

INTRODUCTION TO OPENACC

Lecture 1: Analyzing and Parallelizing with OpenACC, October 26, 2016





Course Objective:

Enable *you* to to accelerate your applications
with OpenACC.

Course Syllabus

Oct 26: Analyzing and Parallelizing with OpenACC

Nov 2: OpenACC Optimizations

Nov 9: Advanced OpenACC

Recordings:

<https://developer.nvidia.com/intro-to-openacc-course-2016>

ANALYZING AND PARALLELIZING WITH OPENACC

Lecture 1: Jeff Larkin, NVIDIA



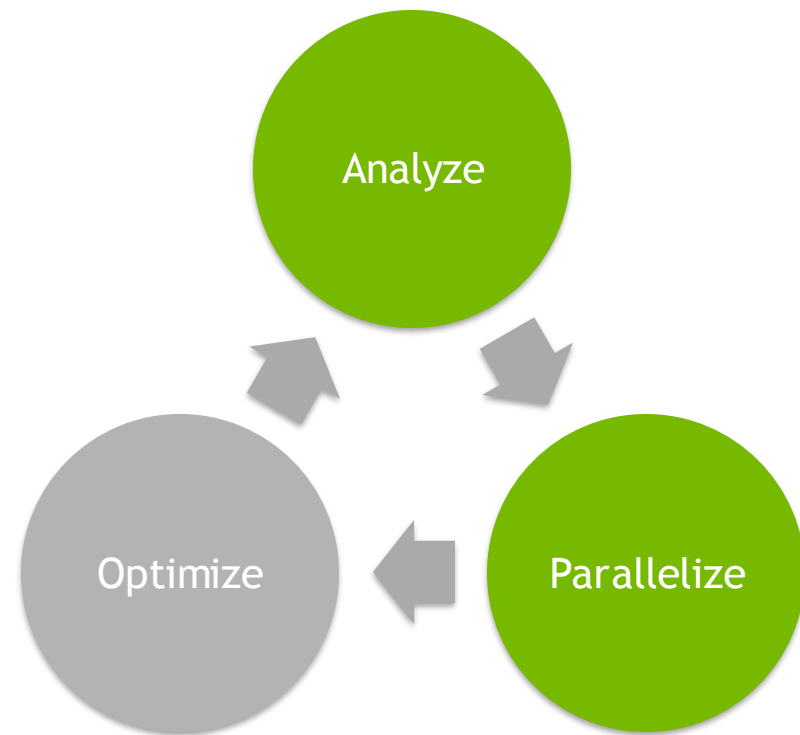
Today's Objectives

Understand what OpenACC is and why to use it

Understand some of the differences between CPU and GPU hardware.

Know how to obtain an application profile using PGProf

Know how to add OpenACC directives to existing loops and build with OpenACC using PGI





Why OpenACC?

OpenACC

Simple | Powerful | Portable

Fueling the Next Wave of
Scientific Discoveries in HPC

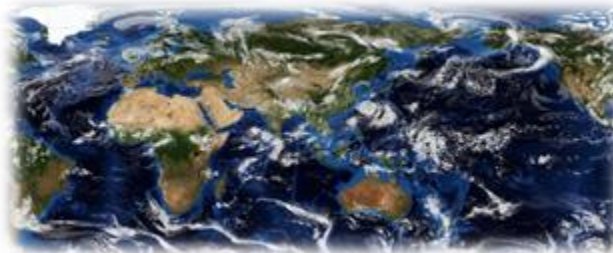
```
main()
{
    <serial code>
    #pragma acc kernels
    //automatically runs on GPU
    {
        <parallel code>
    }
}
```

University of Illinois
PowerGrid- MRI Reconstruction



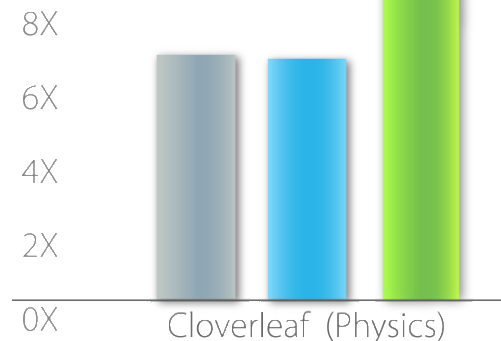
70x Speed-Up
2 Days of Effort

RIKEN Japan
NICAM- Climate Modeling



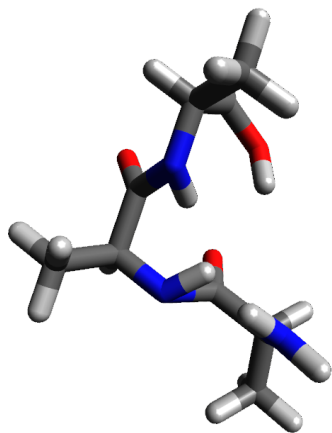
7-8x Speed-Up
5% of Code Modified

■ CPU: OpenMP
■ CPU: OpenACC
■ GPU: OpenACC



LS-DALTON

Large-scale application for calculating high-accuracy molecular energies



“OpenACC makes GPU computing approachable for domain scientists. Initial OpenACC implementation required only minor effort, and more importantly, *no modifications* of our existing CPU implementation.”

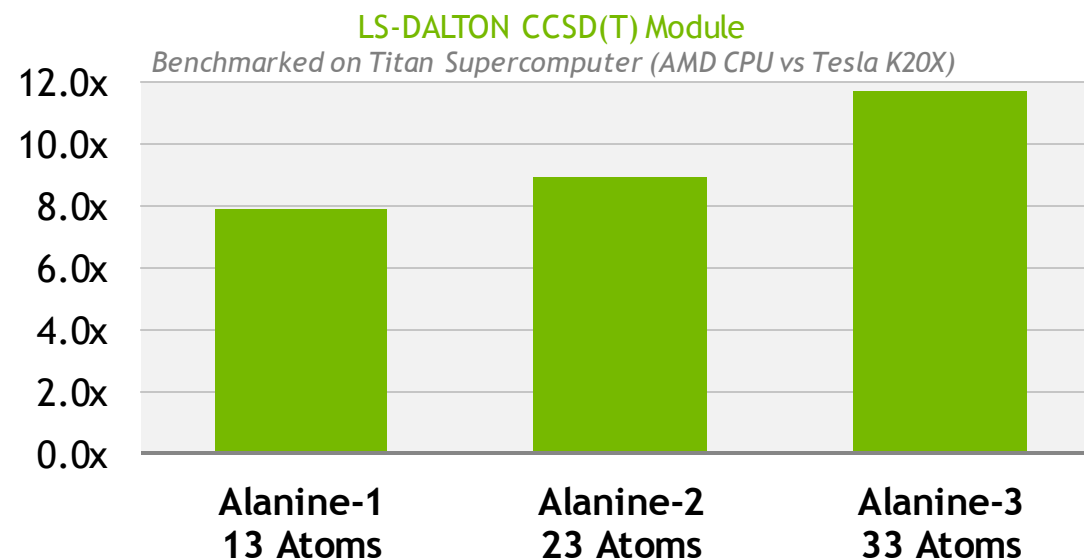
Janus Juul Eriksen, PhD Fellow
qLEAP Center for Theoretical Chemistry, Aarhus University



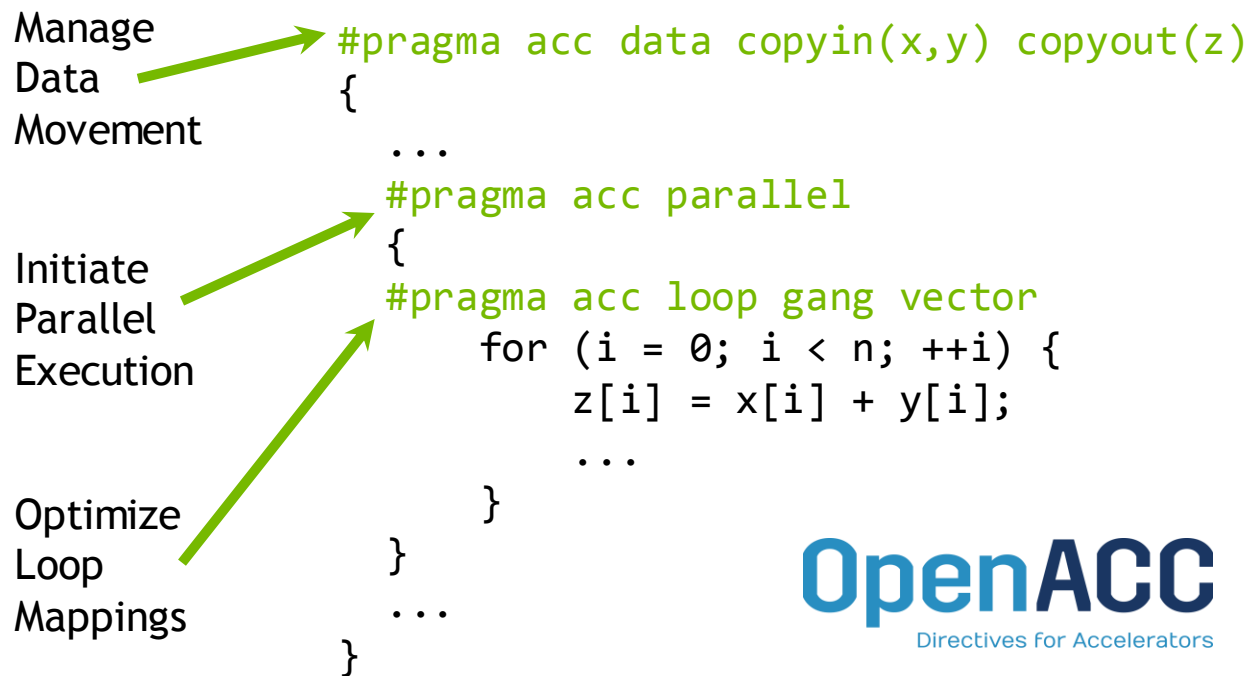
Minimal Effort

Lines of Code Modified	# of Weeks Required	# of Codes to Maintain
<100 Lines	1 Week	1 Source

Big Performance



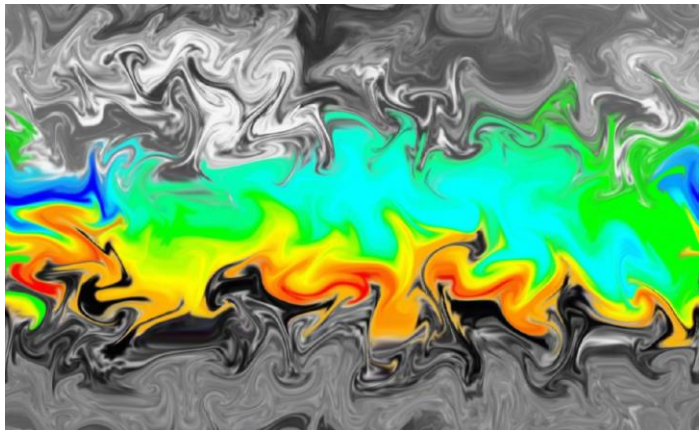
OpenACC Directives



- Incremental
- Single source
- Interoperable
- Performance portable

OpenACC Performance Portability: CloverLeaf

Hydrodynamics Application

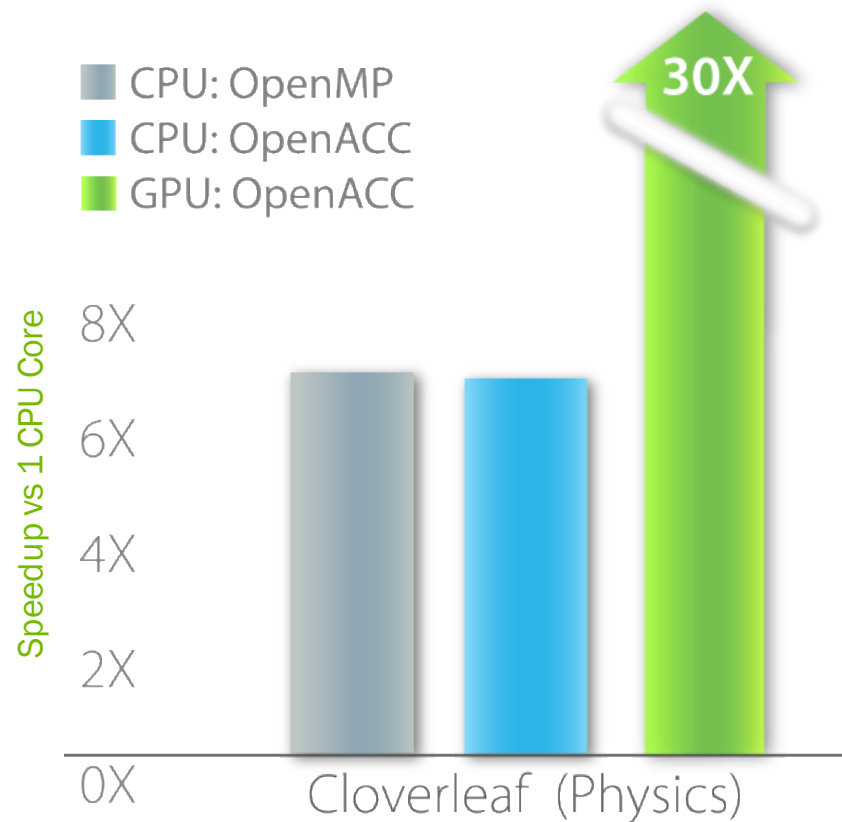


“We were extremely impressed that we can run OpenACC on a CPU with *no code change* and get *equivalent performance* to our OpenMP/MPI implementation.”

Wayne Gaudin and Oliver Perks
Atomic Weapons Establishment, UK

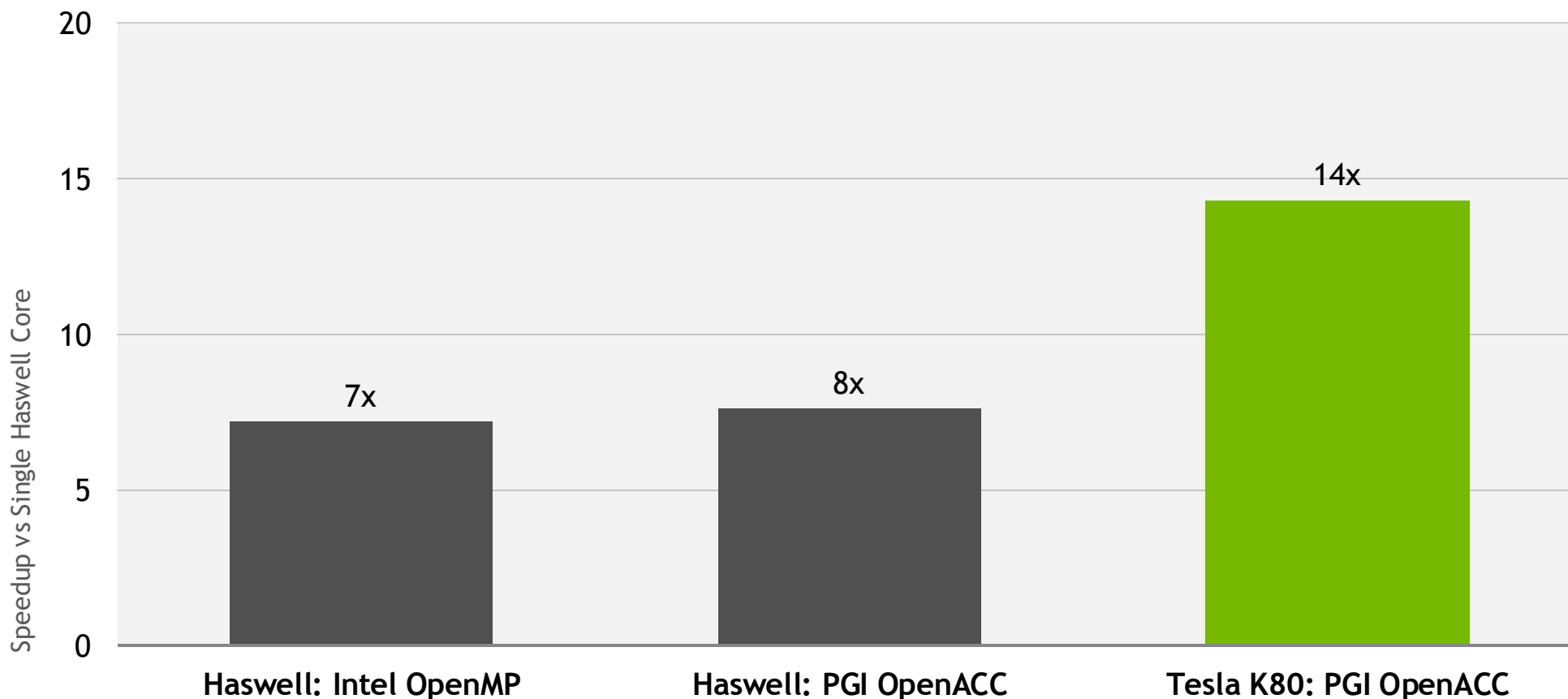


OpenACC Performance Portability



Benchmarked Intel(R) Xeon(R) CPU E5-2690 v2 @ 3.00GHz, Accelerator: Tesla K80

CloverLeaf on Dual Haswell vs Tesla K80

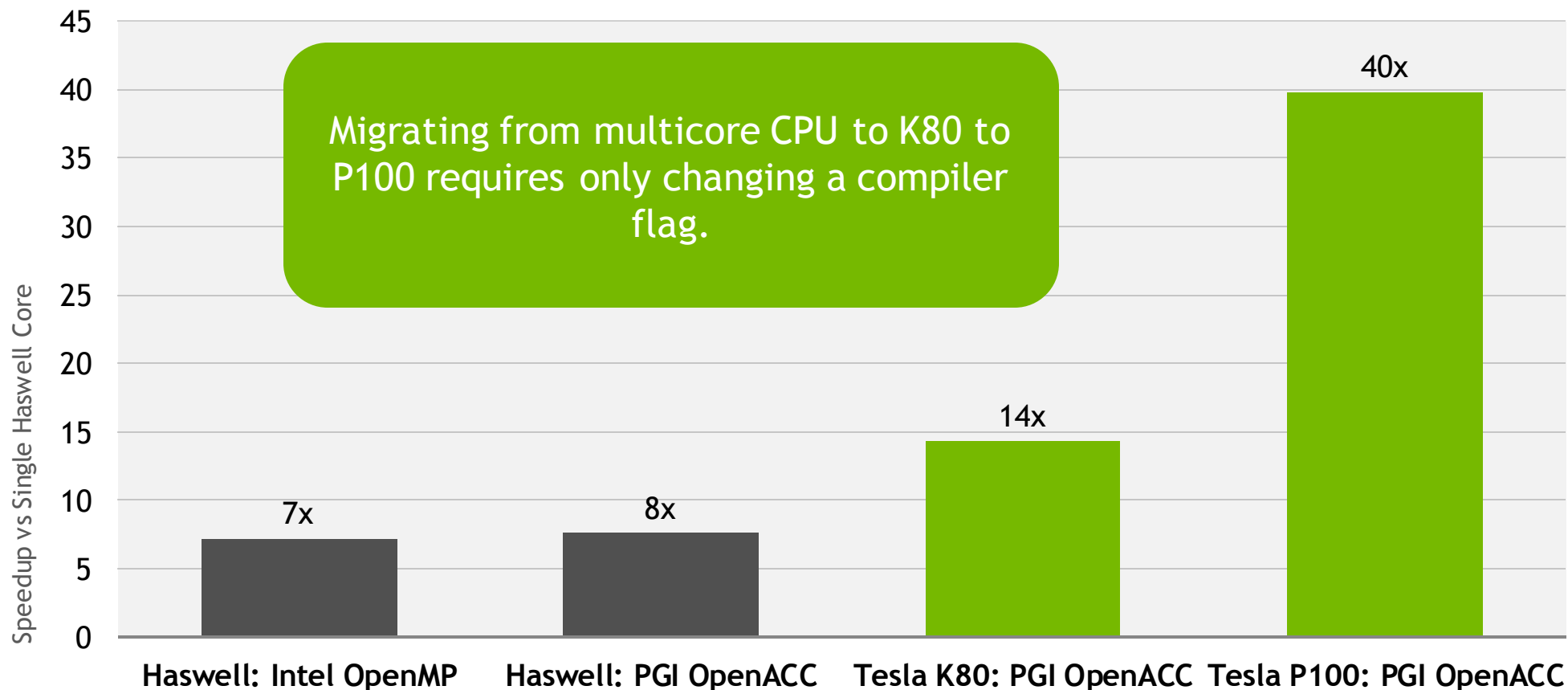


CPU: Intel Xeon E5-2698 v3, 2 sockets, 32 cores, 2.30 GHz, HT disabled

GPU: NVIDIA Tesla K80 (single GPU)

OS: CentOS 6.6, Compiler: PGI 16.5

CloverLeaf on Tesla P100 Pascal

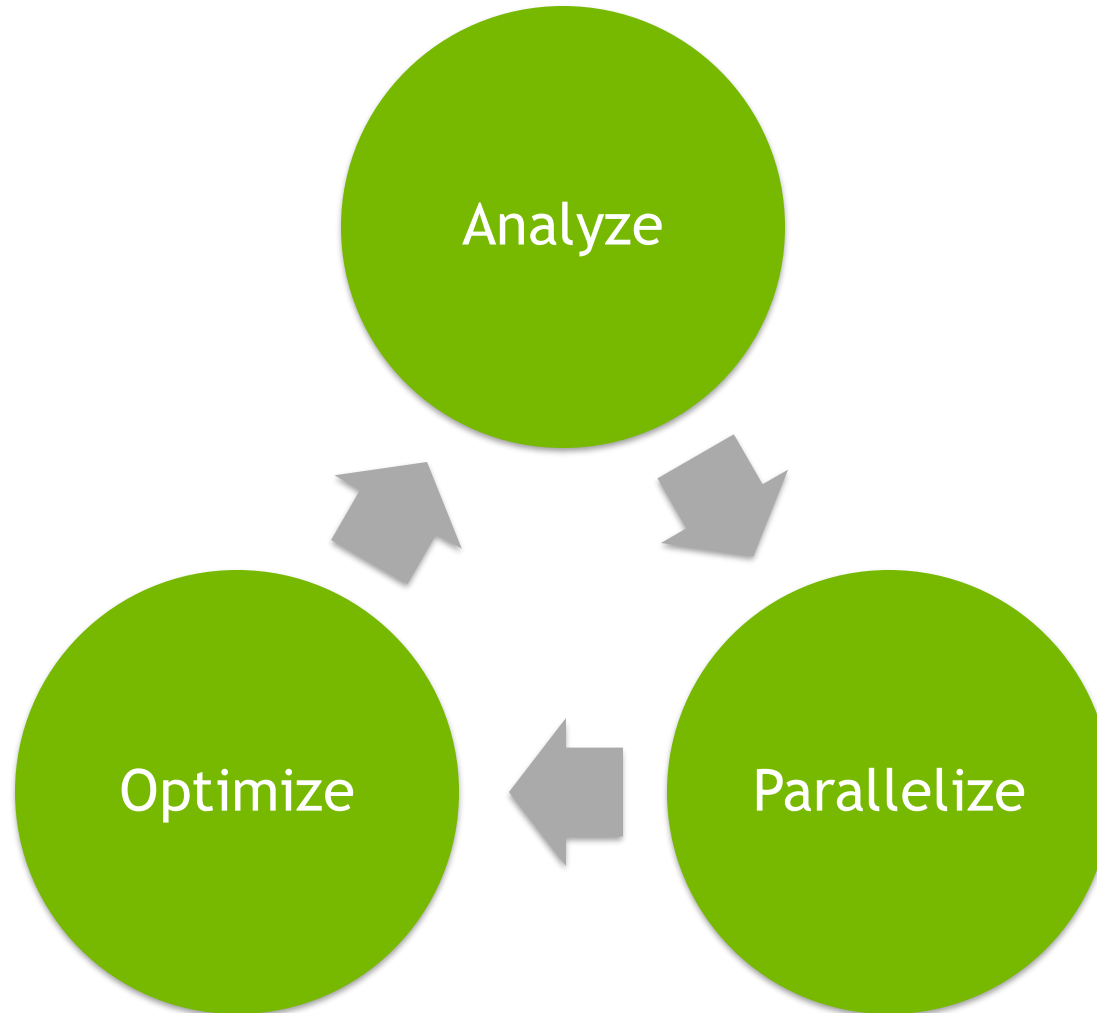


CPU: Intel Xeon E5-2698 v3, 2 sockets, 32 cores, 2.30 GHz, HT disabled

GPU: NVIDIA Tesla K80 (single GPU), NVIDIA Tesla P100 (Single GPU)

OS: CentOS 6.6, Compiler: PGI 16.5

3 Steps to Accelerate with OpenACC



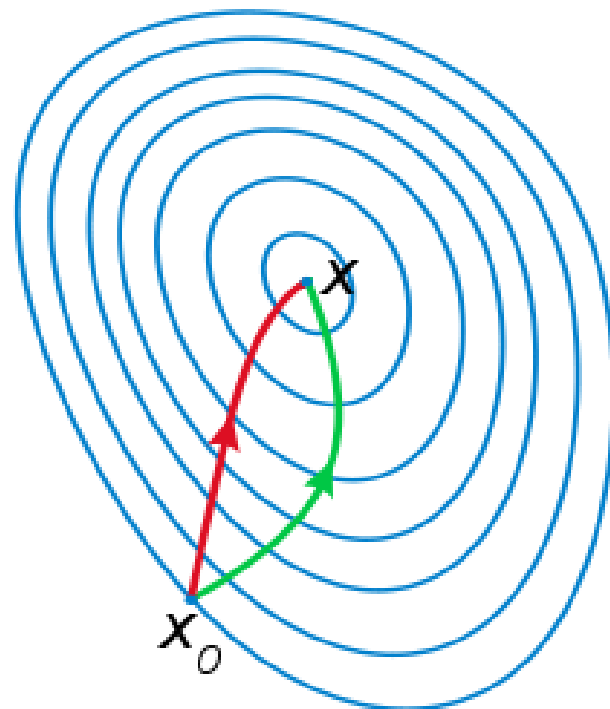
Case Study: Conjugate Gradient

A sample code implementing the conjugate gradient method has been provided in C/C++ and Fortran.

- *To save space, only the C will be shown in slides.*

You do not need to understand the algorithm to proceed, but should be able to understand C, C++, or Fortran.

For more information on the CG method, see https://en.wikipedia.org/wiki/Conjugate_gradient_method



Analyze

Analyze

- ▶ Obtain a performance profile
- ▶ Read compiler feedback
- ▶ Understand the code.

The screenshot shows the PGPROF application interface. The top pane displays C code from `matrix_functions.h`. The bottom pane shows a performance profile table with columns for Event, %, and Time.

```
23 unsigned int *row_offsets=A.row_offsets;
24 unsigned int *cols=A.cols;
25 double *Acoefs=A.coefs;
26 double *xcoefs=x.coefs;
27 double *ycoefs=y.coefs;
28
29 for(int i=0;i<num_rows;i++) {
30     double sum=0;
31     int row_start=row_offsets[i];
32     int row_end=row_offsets[i+1];
33     for(int j=row_start;j<row_end;j++) {
34         unsigned int Acol=cols[j];
35         double Acoef=Acoefs[j];
36         double xcoef=xcoefs[Acol];
37         sum+=Acoef*xcoef;
38     }
39     ycoefs[i]=sum;
40 }
41 }
```

Event	%	Time
matvec(matrix const &, vector const &)	76.742%	26.17 s
waxpby(double, vector const &, double const &)	14.819%	5.053 s
dot(vector const &, vector const &)	4.263%	1.454 s
allocate_3d_poisson_matrix(n)	3.881%	1.324 s
_c_mset8	0.206%	0.07 s
munmap	0.088%	0.03 s

Obtain a Profile

A application profile helps to understand where time is spent

What routines are *hotspots*?

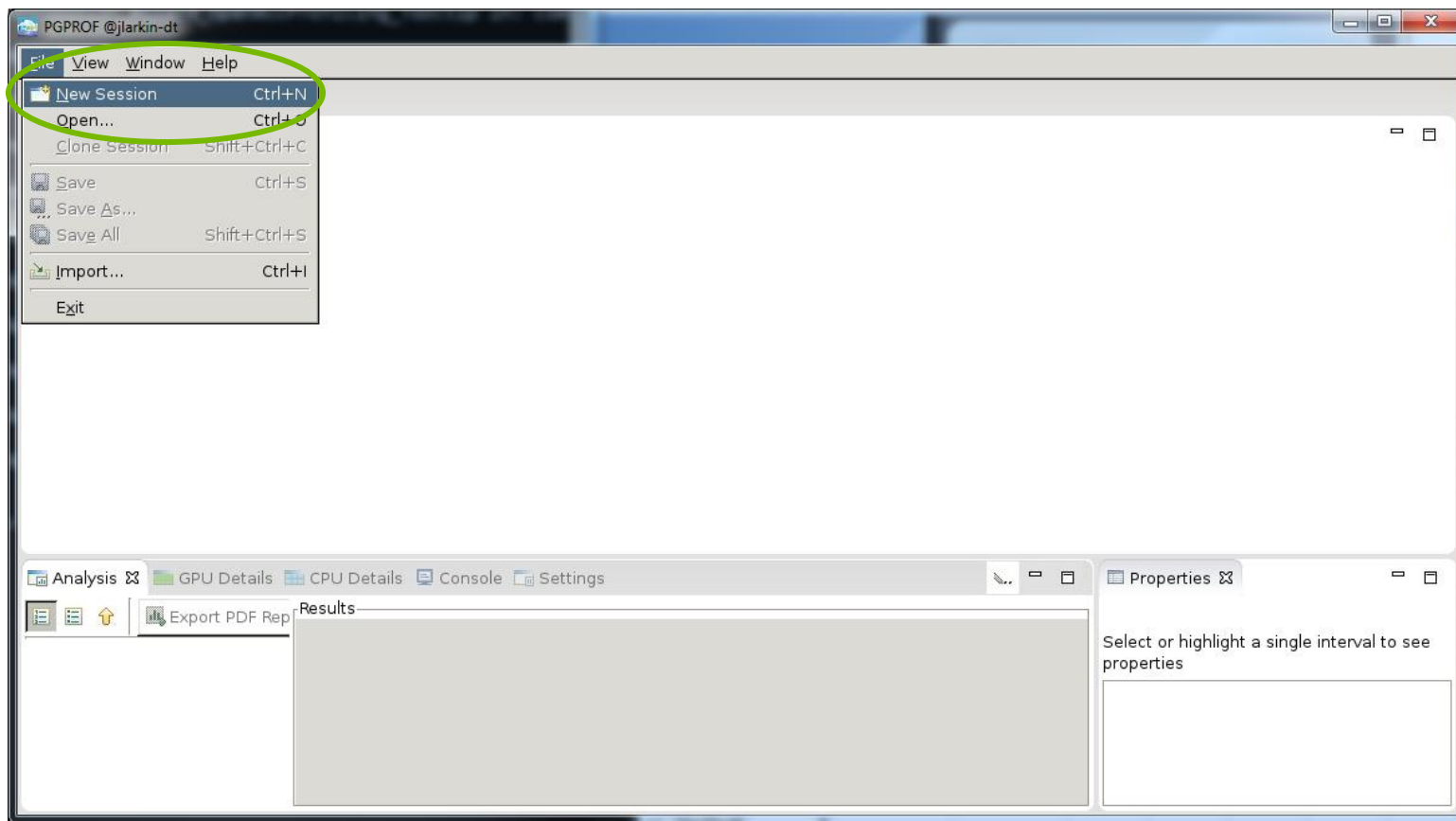
Focusing on the hotspots delivers the greatest performance impact

A variety of profiling tools are available: gprof, nvprof, CrayPAT, TAU, Vampir

We'll use PGProf, which comes with the PGI compiler

```
$ pgprof &
```

PGPROF Profiler



PGPROF Profiler

Create New Session @jlarkin-dt

Executable Properties
Set executable properties

Connection: Local Manage connections...

Toolkit: CUDA Toolkit 8.0 (/opt/pgi/linux86-64/2016/cuda/8.0/bin/) Manage...

File: Enter executable file [required] Browse...

Working directory: Enter working directory [optional] Browse...

Arguments: Enter command-line arguments

Profile child processes

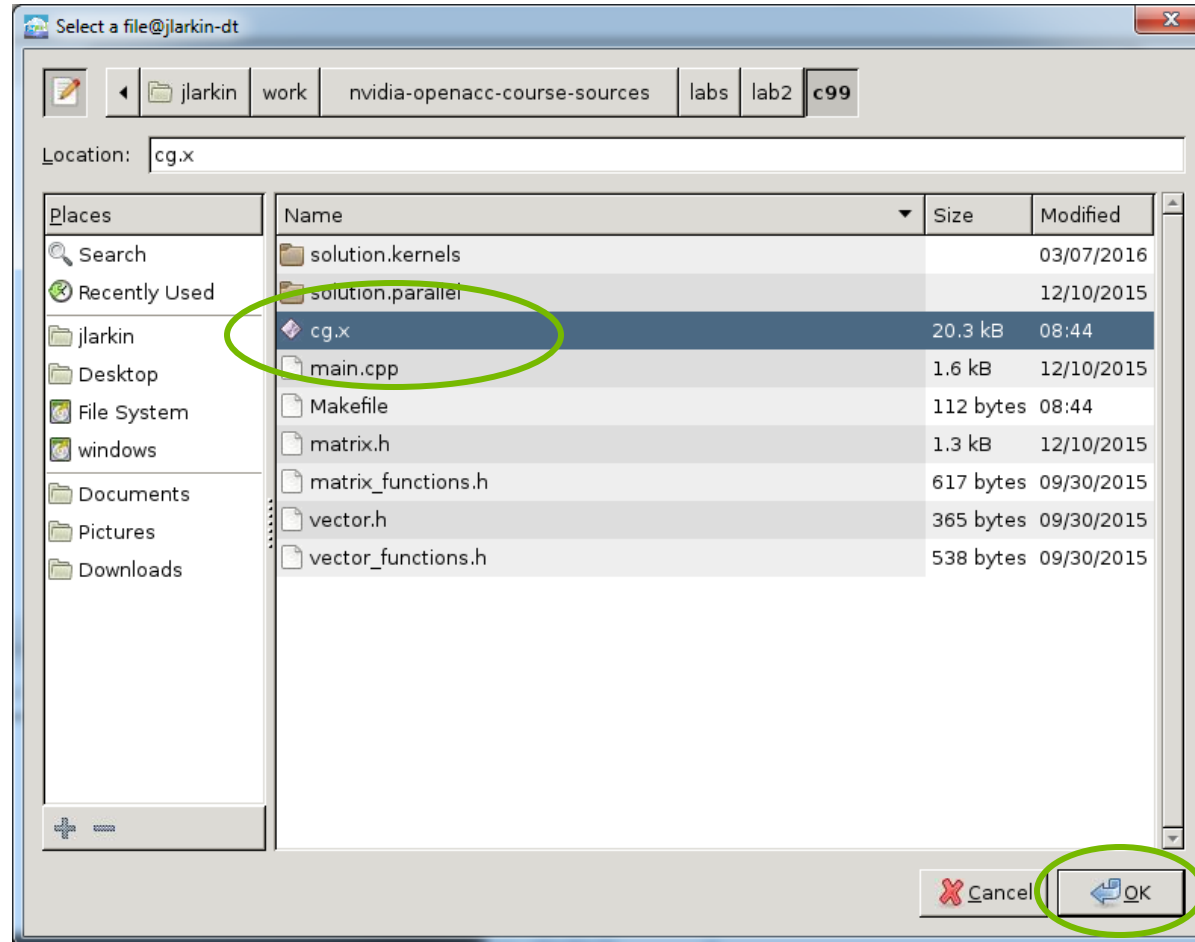
Environment:

Name	Value
------	-------

Add
Delete

< Back Next > Cancel Finish

PGPROF Profiler



PGPROF Profiler

Create New Session @jlarkin-dt

Executable Properties
Set executable properties

Connection: Local Manage connections...

Toolkit: CUDA Toolkit 8.0 (/opt/pgi/linux86-64/2016/cuda/8.0/bin/) Manage...

File: /home/jlarkin/work/nvidia-openacc-course-sources/labs/lab2/c99/cg.x Browse...

Working directory: Enter working directory [optional] Browse...

Arguments: Enter command-line arguments

Profile child processes

Environment:

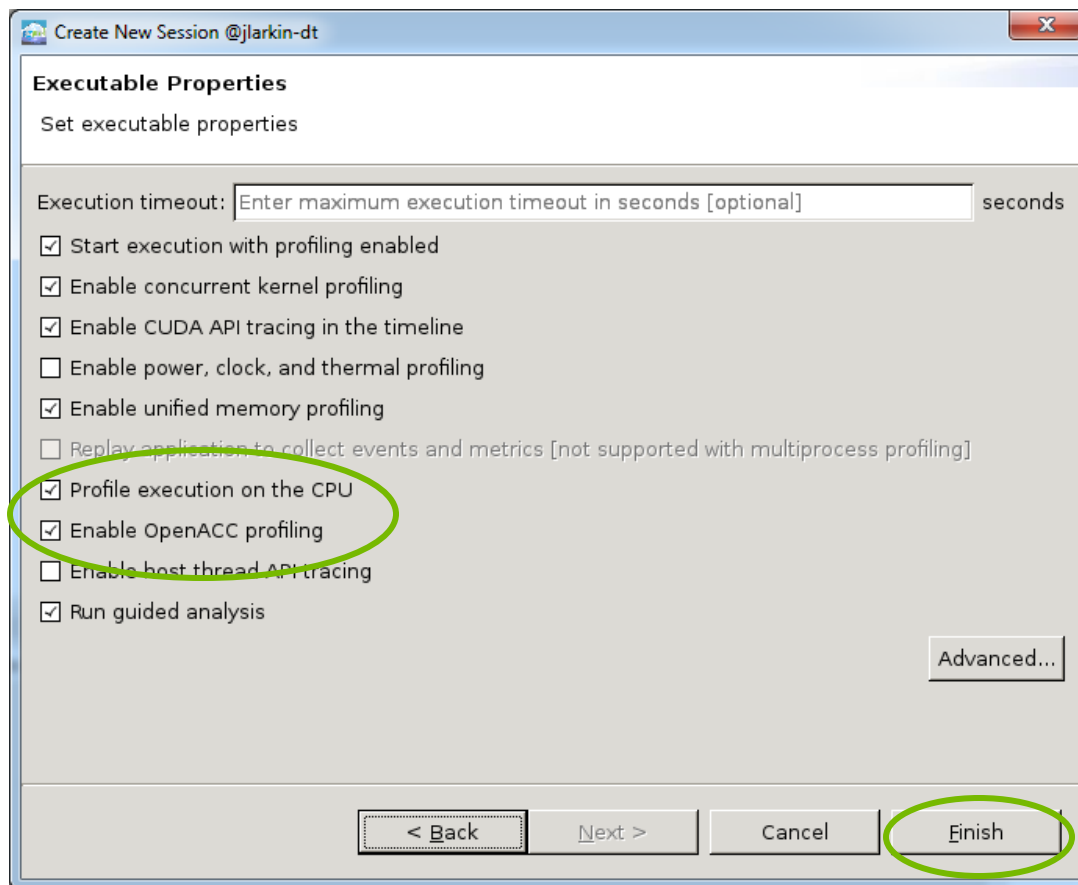
Name	Value
------	-------

Add

Delete

< Back Next > Cancel Finish

PGPROF Profiler



The image shows a screenshot of the 'Create New Session' dialog box in the PGPROF Profiler. The dialog has a title bar that reads 'Create New Session @jlarkin-dt'. The main content area is titled 'Executable Properties' and contains the instruction 'Set executable properties'. Below this, there is a text input field for 'Execution timeout:' with the placeholder text 'Enter maximum execution timeout in seconds [optional]' and a 'seconds' label. A list of checkboxes follows, with the following options checked: 'Start execution with profiling enabled', 'Enable concurrent kernel profiling', 'Enable CUDA API tracing in the timeline', 'Enable unified memory profiling', 'Profile execution on the CPU', 'Enable OpenACC profiling', and 'Run guided analysis'. The options 'Enable power, clock, and thermal profiling' and 'Replay application to collect events and metrics [not supported with multiprocess profiling]' are unchecked. The 'Enable host thread API tracing' option is also unchecked. A green oval highlights the 'Profile execution on the CPU' and 'Enable OpenACC profiling' options. At the bottom right of the main area is an 'Advanced...' button. The bottom of the dialog features four buttons: '< Back', 'Next >', 'Cancel', and 'Finish'. The 'Finish' button is highlighted with a green oval.

Create New Session @jlarkin-dt

Executable Properties
Set executable properties

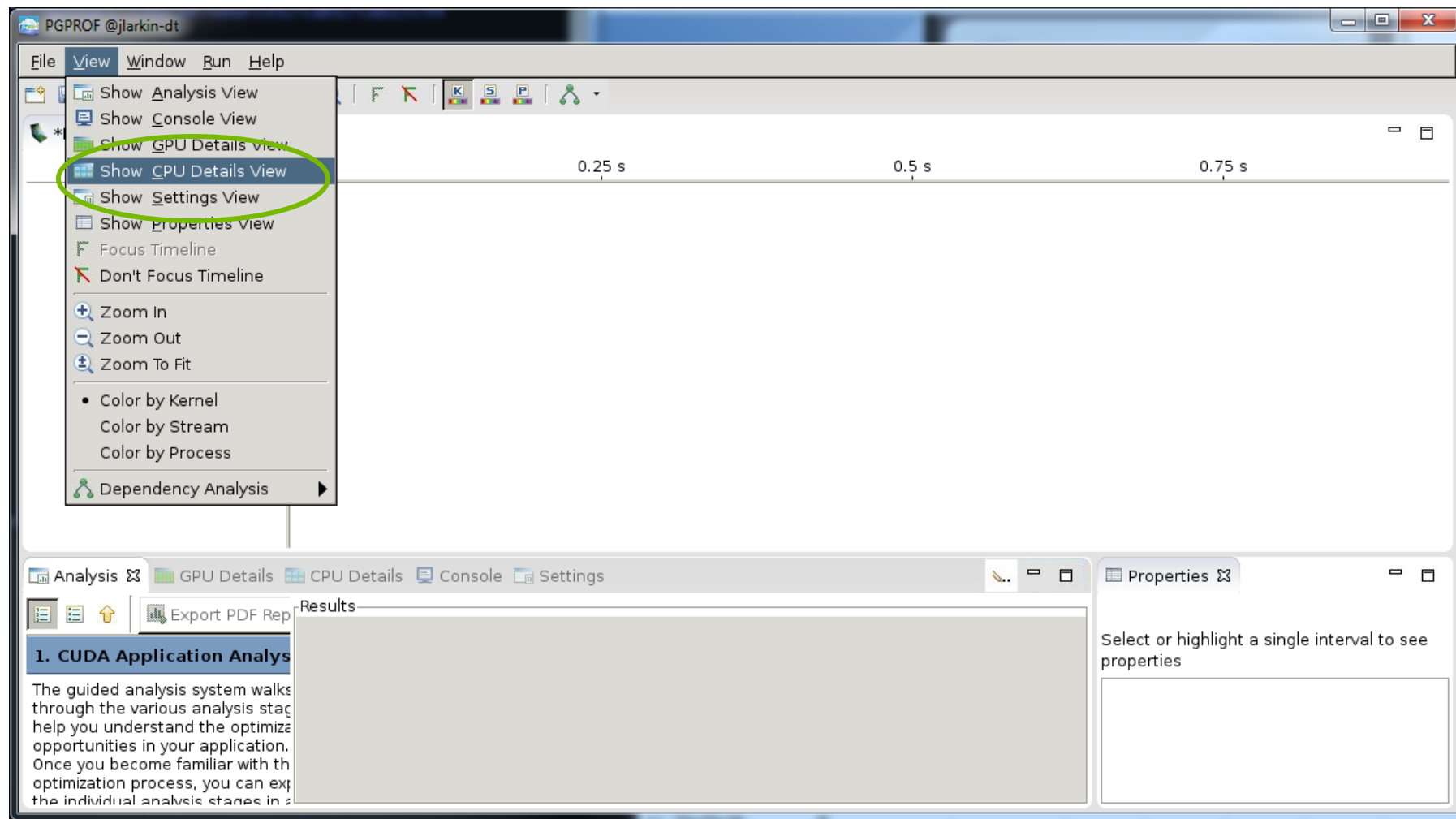
Execution timeout: seconds

- ☒ Start execution with profiling enabled
- ☒ Enable concurrent kernel profiling
- ☒ Enable CUDA API tracing in the timeline
- ☐ Enable power, clock, and thermal profiling
- ☒ Enable unified memory profiling
- ☐ Replay application to collect events and metrics [not supported with multiprocess profiling]
- ☒ Profile execution on the CPU
- ☒ Enable OpenACC profiling
- ☐ Enable host thread API tracing
- ☒ Run guided analysis

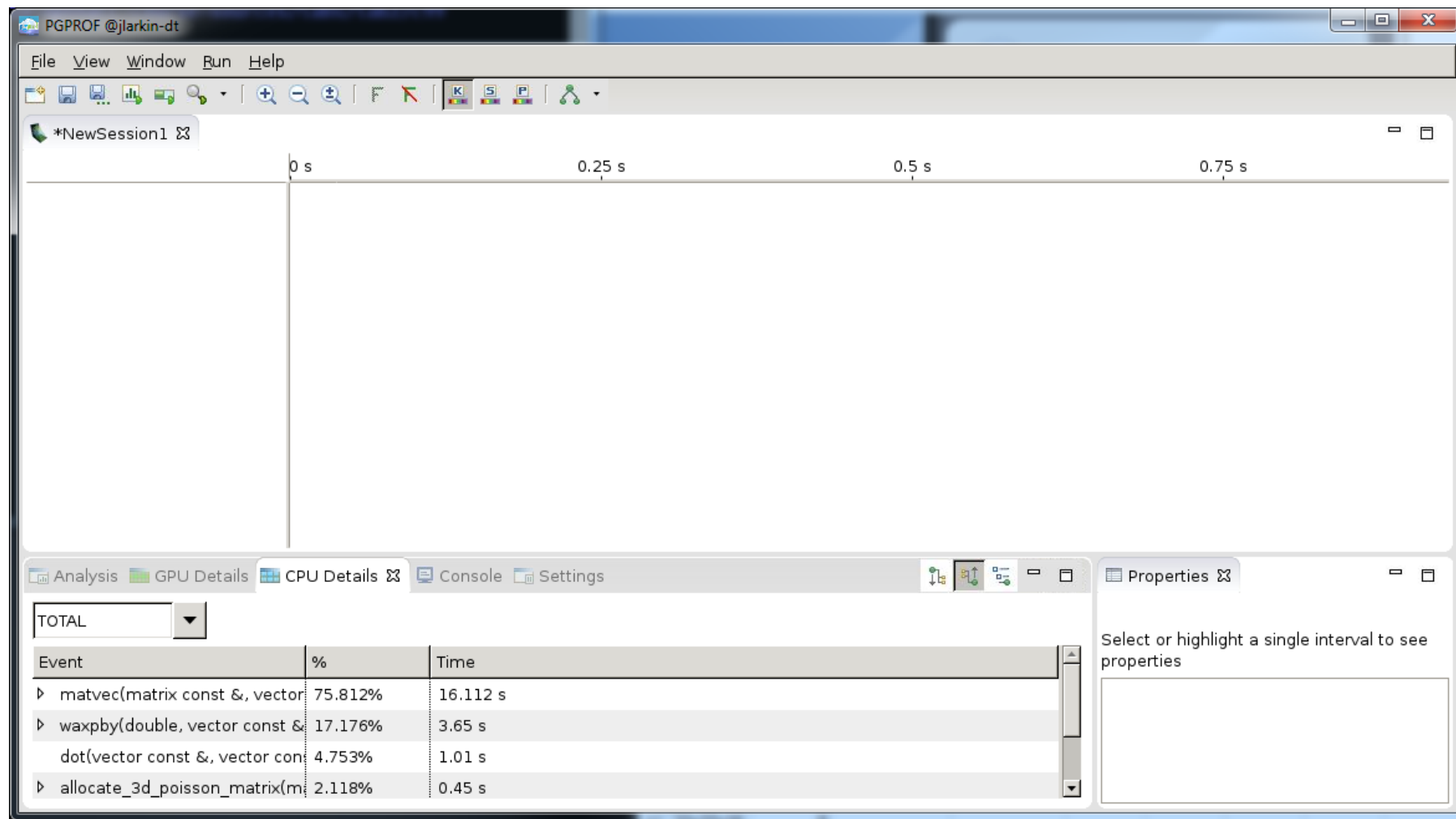
Advanced...

< Back Next > Cancel Finish

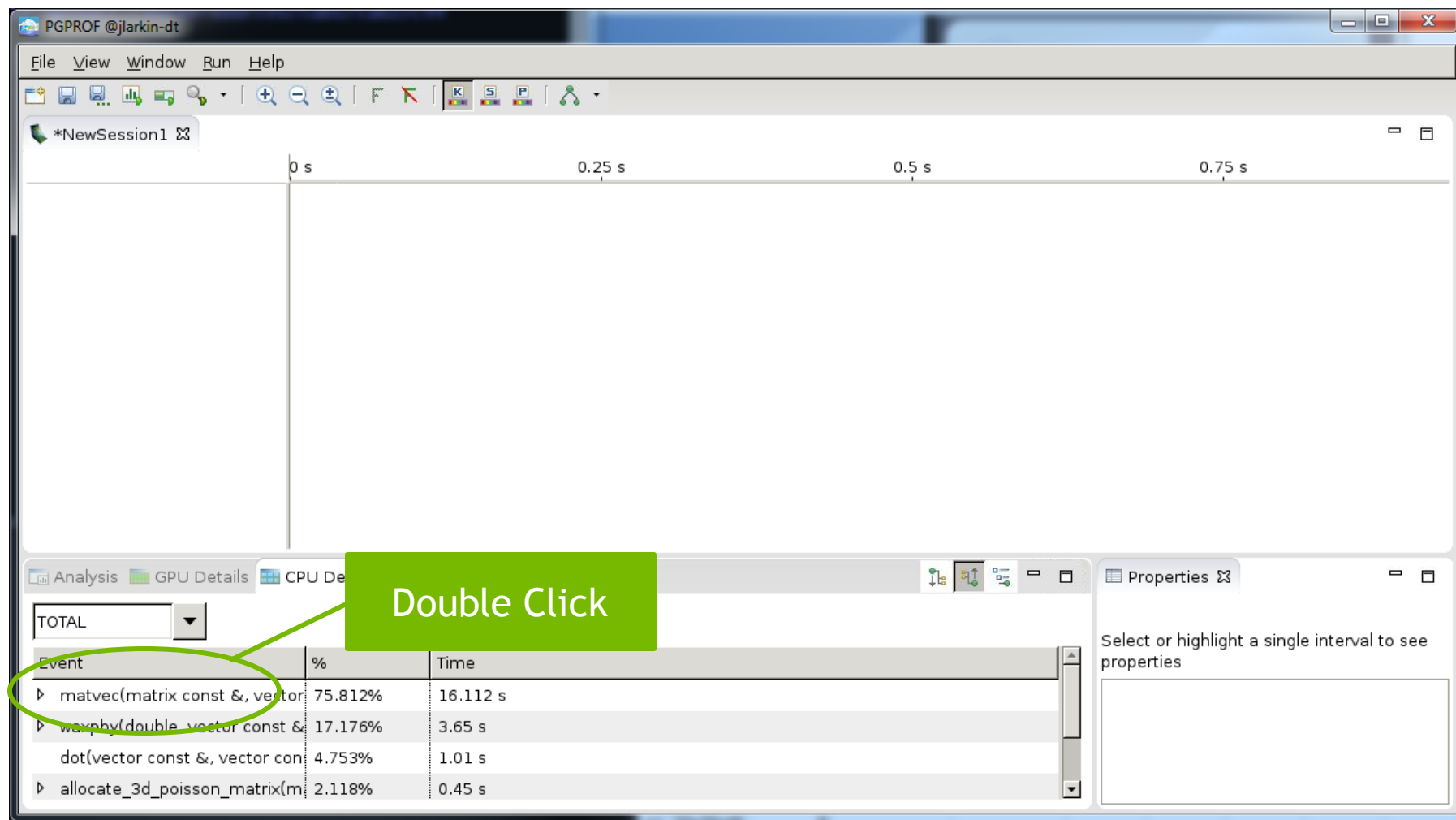
PGPROF Profiler



PGPROF Profiler



PGPROF Profiler



PGPROF Profiler

The screenshot displays the PGPROF Profiler application window. The main editor shows a C++ file named `matrix_functions.h` with the following code:

```
8 unsigned int *row_offsets=A.row_offsets;
9 unsigned int *cols=A.cols;
10 double *Acoefs=A.coefs;
11 double *xcoefs=x.coefs;
12 double *ycoefs=y.coefs;
13
14 for(int i=0;i<num_rows;i++) {
15     double sum=0;
16     int row_start=row_offsets[i];
17     int row_end=row_offsets[i+1];
18     for(int j=row_start;j<row_end;j++) {
19         unsigned int Acol=cols[j];
20         double Acoef=Acoefs[j];
21         double xcoef=xcoefs[Acol];
22         sum+=Acoef*xcoef;
23     }
24     ycoefs[i]=sum;
25 }
```

The bottom panel shows the 'CPU Details' tab with a table of performance data. The table has columns for 'Event', '%', and 'Time'. The data is as follows:

Event	%	Time
matvec(matrix const &, vector const &)	75.812%	16.112 s
waxpby(double, vector const &, vector const &)	17.176%	3.65 s
dot(vector const &, vector const &)	4.753%	1.01 s
allocate_3d_poisson_matrix(m, n, k)	2.118%	0.45 s

The right sidebar contains a 'Properties' tab with the instruction: 'Select or highlight a single interval to see properties'.

Compiler Feedback

- ▶ Before we can make changes to the code, we need to understand how the compiler is optimizing
- ▶ With PGI, this can be done with the `-Minfo` and `-Mneginfo` flags

```
matvec(const matrix &, const vector &, const  
vector &):  
    23, include "matrix_functions.h"  
        Generated 2 alternate versions of the loop  
        Generated vector sse code for the loop  
        Generated 2 prefetch instructions for the  
loop
```

```
$ pgc++ -Minfo=all,ccff -Mneginfo
```

Compiler Feedback in PGProf

The screenshot displays the PGProf application interface. The main window shows a C source file named `matrix_functions.h` with the following code:

```
8 unsigned int *row_offsets=A.row_offsets;
9 unsigned int *cols=A.cols;
10 double *Acoefs=A.coefs;
11 double *xcoefs=x.coefs;
12 double *ycoefs=y.coefs;
13
14 for(int i=0;i<num_rows;i++) {
15     double sum=0;
16     int row_start=row_offsets[i];
17     int row_end=row_offsets[i+1];
18
19     // Multiple markers at this line
20     // - Generated vector sse code for the loop
21     // - Intensity = 1.00
22     // - Generated 2 prefetch instructions for the loop
23     // - Generated 2 alternate versions of the loop
24     ycoefs[i]=sum;
25 }
```

A tooltip is visible over line 18, indicating multiple markers and listing generated optimizations: vector sse code, intensity of 1.00, prefetch instructions, and alternate loop versions.

The bottom panel shows the 'Analysis' tab with a table of performance data:

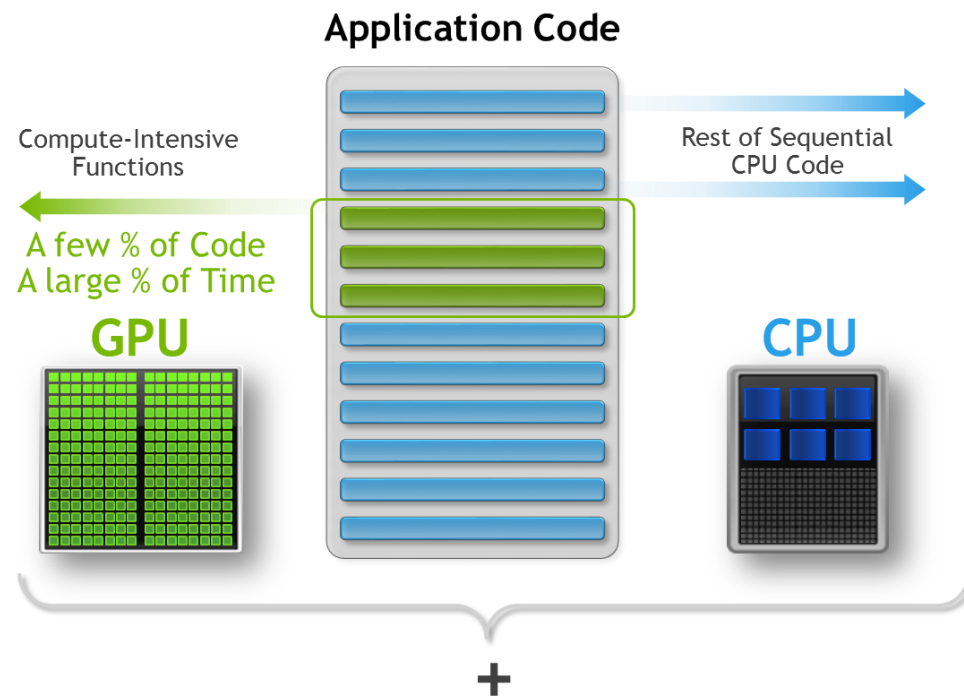
Event	%	Time
matvec(matrix const &, vector const &)	75.812%	16.112 s
waxpby(double, vector const &, vector const &)	17.176%	3.65 s
dot(vector const &, vector const &)	4.753%	1.01 s
allocate_3d_poisson_matrix(m, n, k)	2.118%	0.45 s

The right sidebar contains a 'Properties' panel with the instruction: 'Select or highlight a single interval to see properties'.

Parallelize

Parallelize

- ▶ Insert OpenACC directives around important loops
- ▶ Enable OpenACC in the compiler
- ▶ Run on a parallel platform

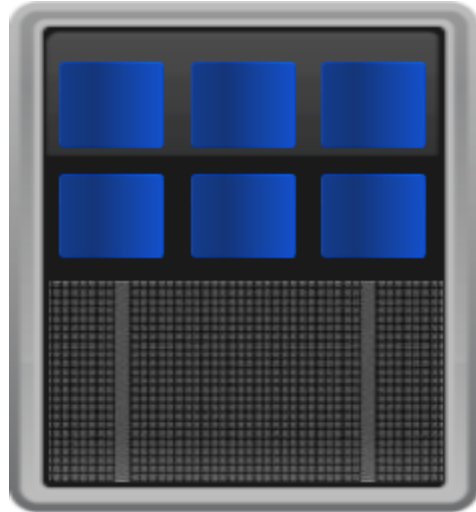


Accelerated Computing

10x Performance & 5x Energy Efficiency for HPC

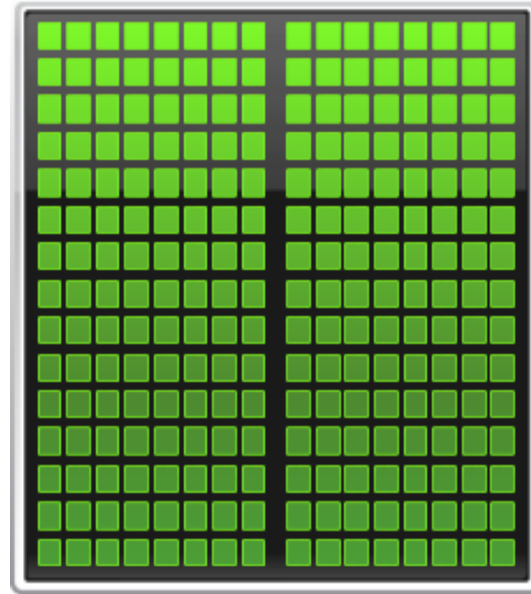
CPU

Optimized for
Serial Tasks



GPU Accelerator

Optimized for
Parallel Tasks

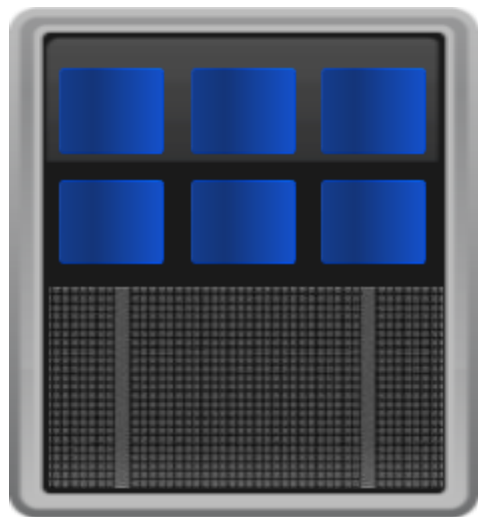


Accelerated Computing

10x Performance & 5x Energy Efficiency for HPC

CPU

Optimized for
Serial Tasks



CPU Strengths

- Very large main memory
- Very fast clock speeds
- Latency optimized via large caches
- Small number of threads can run very quickly

CPU Weaknesses

- Relatively low memory bandwidth
- Cache misses very costly
- Low performance/watt

Accelerated Computing

10x Performance & 5x Energy Efficiency for HPC

GPU Strengths

- High bandwidth main memory
- Significantly more compute resources
- Latency tolerant via parallelism
- High throughput
- High performance/watt

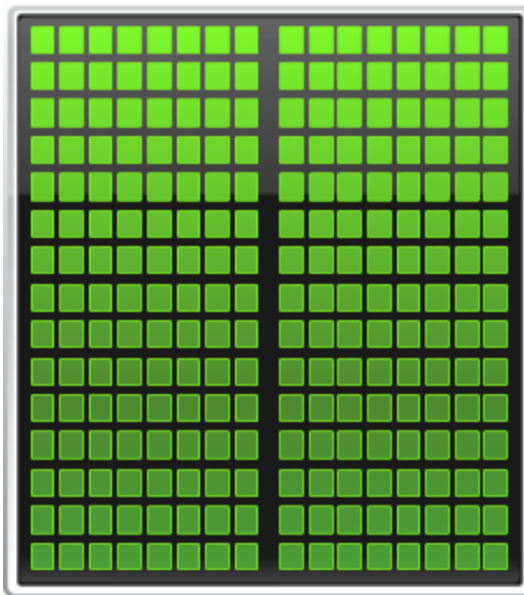
GPU Weaknesses

- Relatively low memory capacity
- Low per-thread performance



GPU Accelerator

Optimized for
Parallel Tasks



Speed v. Throughput

Speed

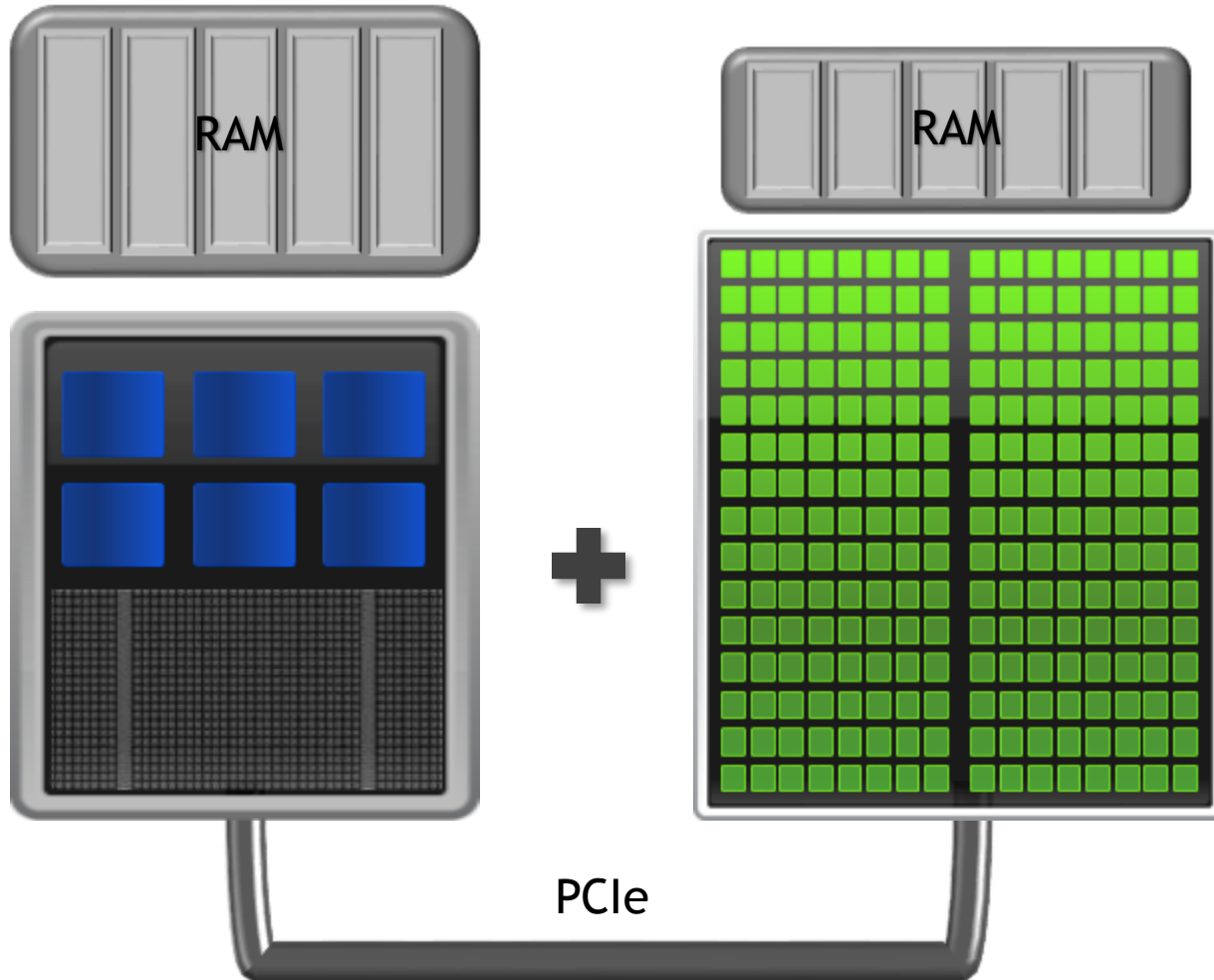


Throughput



Which is better depends on your needs...

Accelerator Nodes



CPU and GPU communicate via PCIe

- Data must be copied between these memories over PCIe
- PCIe Bandwidth is much lower than either memories

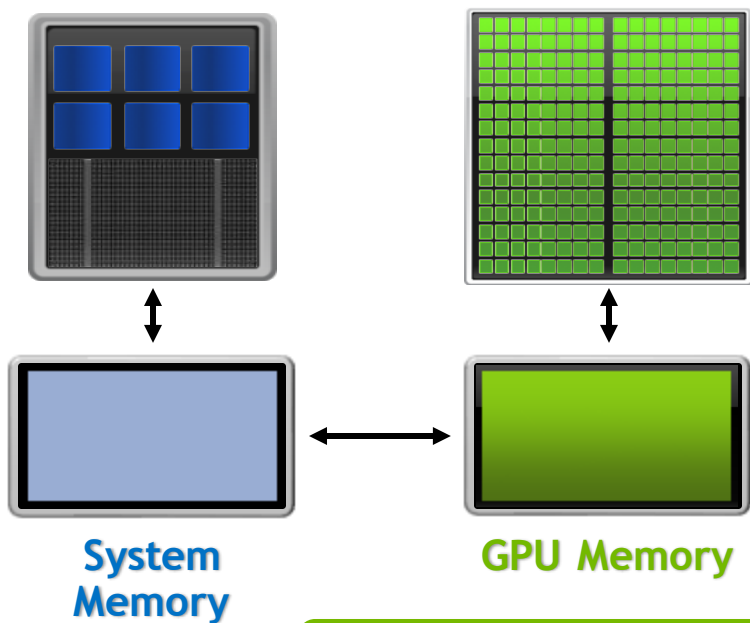
Obtaining high performance on GPU nodes often requires reducing PCIe copies to a minimum

CUDA Unified Memory

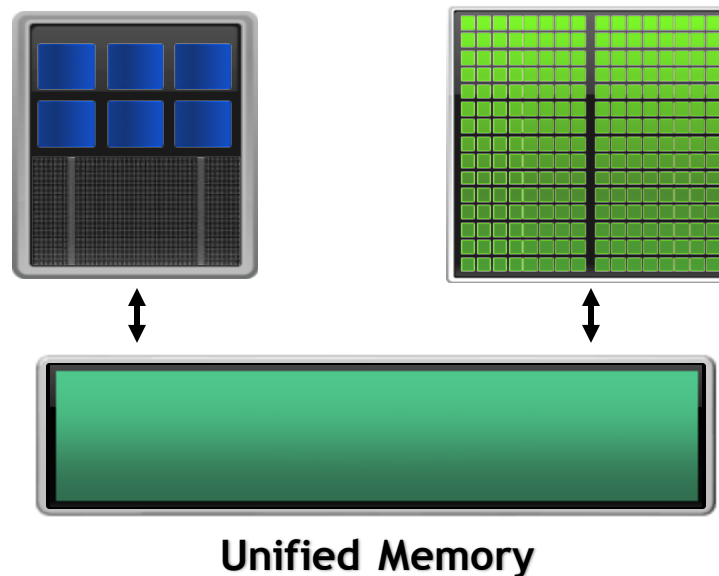
Simplified Developer Effort

Sometimes referred to as
“managed memory.”

Without Unified Memory



With Unified Memory



New “Pascal” GPUs handle Unified Memory in hardware.

OpenACC Parallel Directive

Generates parallelism

```
#pragma acc parallel
```

```
{
```

When encountering the *parallel* directive, the compiler will generate *1 or more parallel gangs*, which execute redundantly.

```
}
```

OpenACC Parallel Directive

Generates parallelism

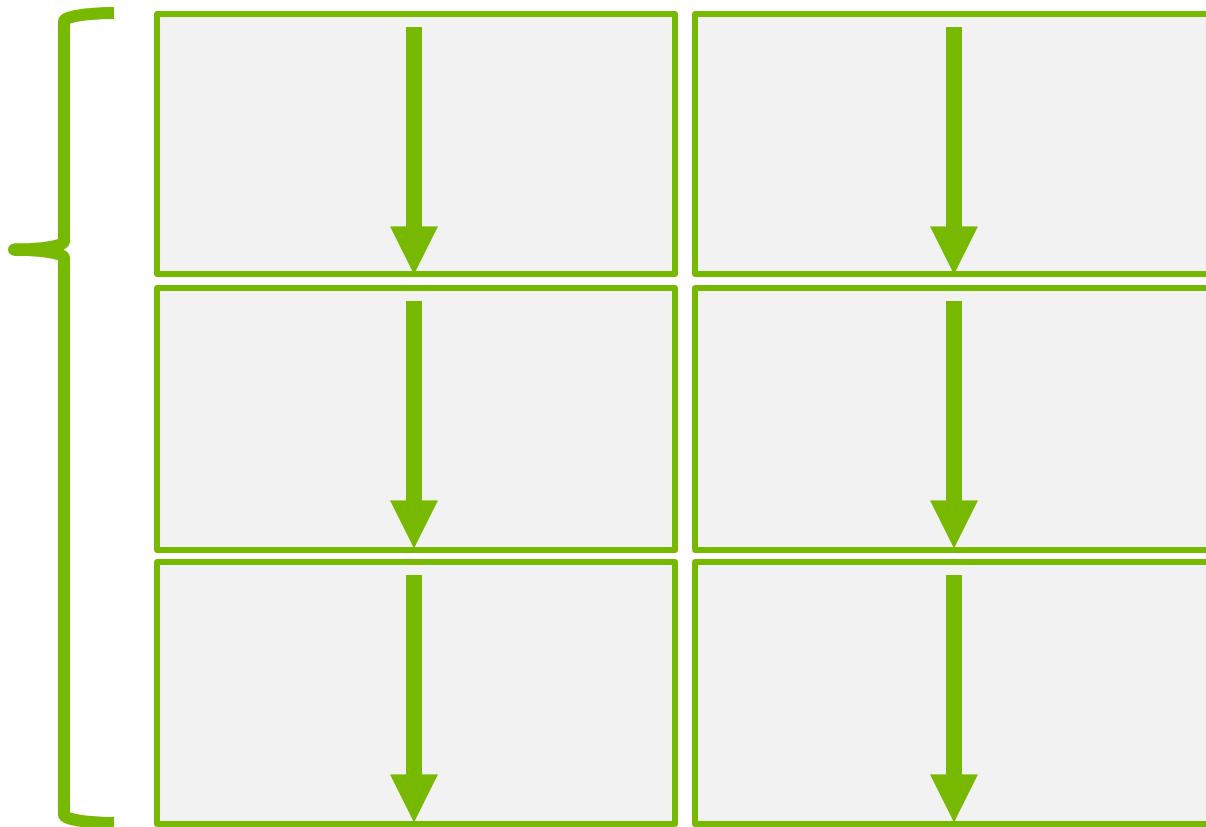
```
#pragma acc parallel
```

```
{
```



```
}
```

When encountering the *parallel* directive, the compiler will generate *1 or more parallel gangs*, which execute redundantly.



OpenACC Loop Directive

Identifies loops to run in parallel

```
#pragma acc parallel
```

```
{
```



```
#pragma acc loop
```

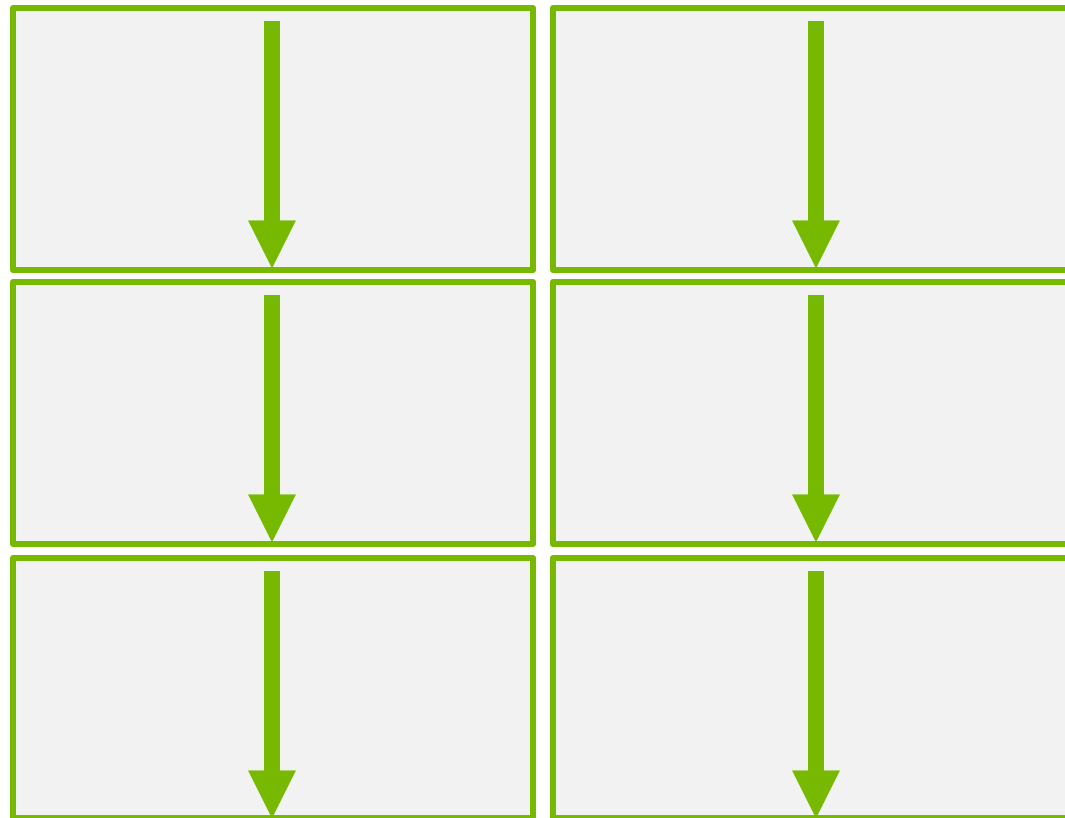
```
for (i=0;i<N;i++)
```

```
{
```

The *loop* directive
informs the compiler
which loops to
parallelize.

```
}
```

```
}
```



OpenACC Loop Directive

Identifies loops to run in parallel

```
#pragma acc parallel
```

```
{
```

```
    #pragma acc loop
```

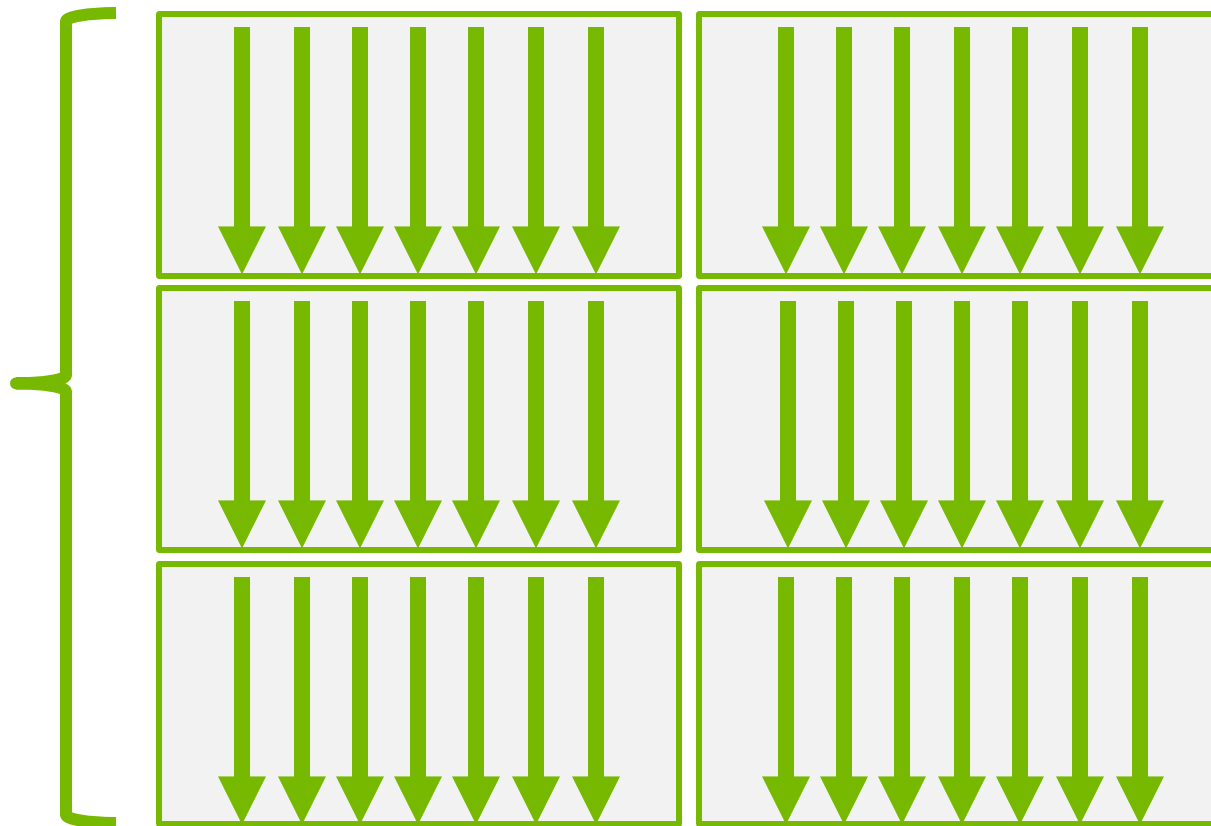
```
    for (i=0; i<N; i++)
```

```
    {
```

The *loop* directive
informs the compiler
which loops to
parallelize.

```
    }
```

```
}
```



OpenACC Parallel Loop Directive

Generates parallelism and identifies loop in one directive

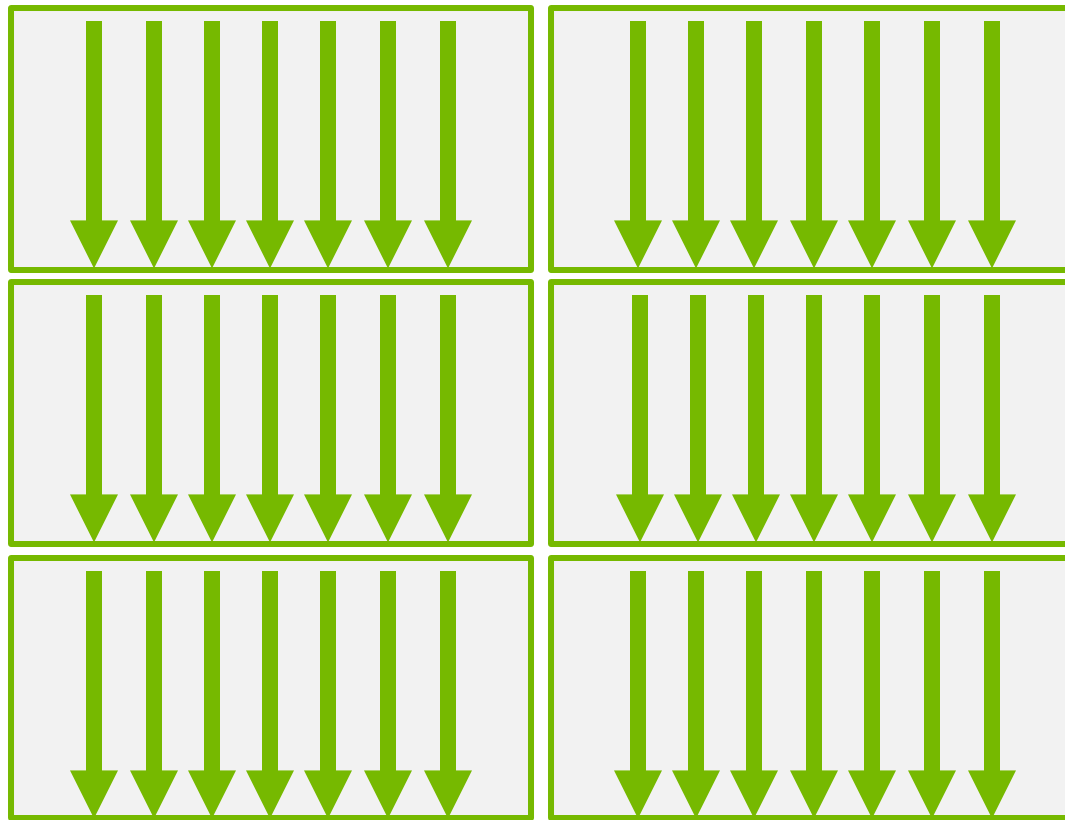
```
#pragma acc parallel loop
```

```
for (i=0;i<N;i++)
```

```
{
```

```
}
```

The *parallel* and *loop* directives are frequently combined into one.



Case Study: Parallelize

Normally we would start with the most time-consuming routine to deliver the greatest performance impact.

In order to ease you in to writing parallel code, I will instead start with the simplest routine.

Parallelize Waxpby

```
void waxpby(...) {  
  
    #pragma acc parallel loop  
    for(int i=0;i<n;i++) {  
        wcoefs[i] =  
            alpha*xcoefs[i] +  
            beta*ycoefs[i];  
    }  
}
```

- ▶ Adding a *parallel loop* around the waxpby loop informs the compiler to
 - ▶ Generate parallel gangs on which to execute
 - ▶ Parallelize the loop iterations across the parallel gangs

Build With OpenACC

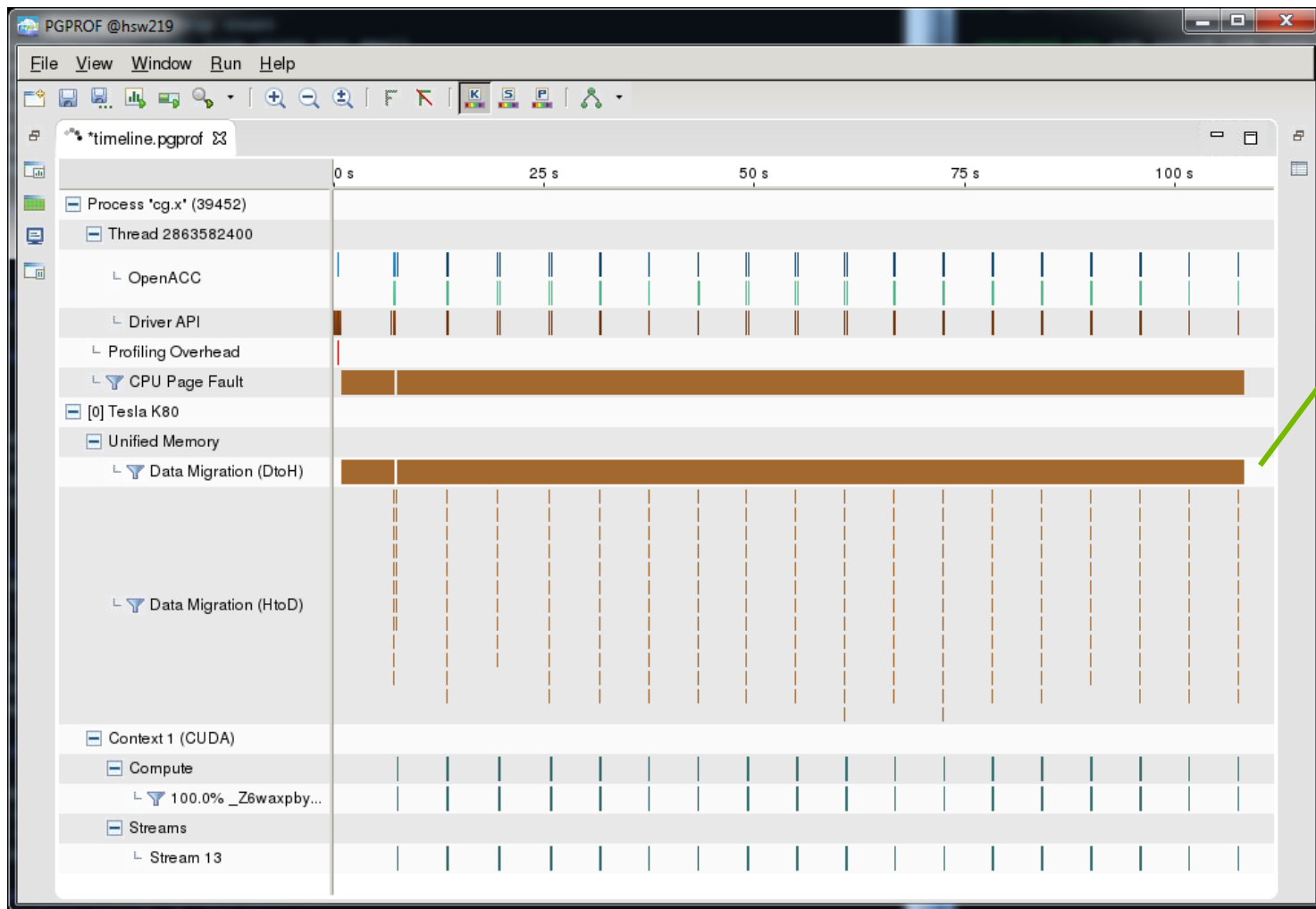
- ▶ The PGI -ta flag enables OpenACC and chooses a *target accelerator*.
- ▶ We'll add the following to our compiler flags:

-ta=tesla:managed

Compiler feedback now:

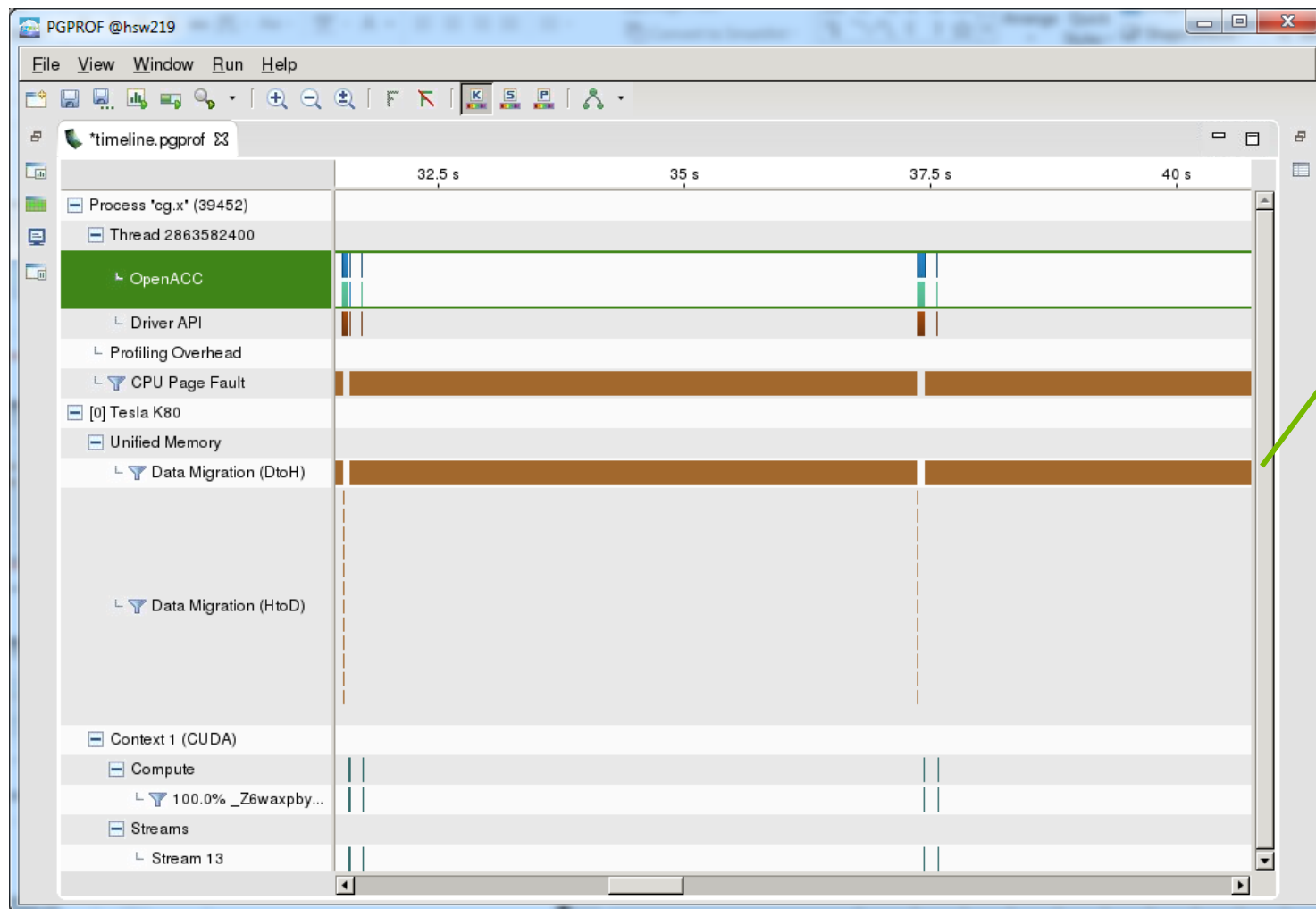
```
waxpby(double, const vector &, double,  
const vector &, const vector &):  
    6, include "vector_functions.h"  
    22, Generating implicit  
copyout(wcoefs[:n])  
        Generating implicit  
copyin(xcoefs[:n],ycoefs[:n])  
        Accelerator kernel generated  
        Generating Tesla code  
    25, #pragma acc loop gang,  
vector(128) /* blockIdx.x threadIdx.x */
```

PGPROF with Parallel waxpby



A significant portion of the time is now spent migrating data between the *host* and *device*.

PGPROF with Parallel waxpby



In order to improve performance, we need to parallelize the remaining functions.

Parallelize Dot

```
double dot(...) {  
  
    #pragma acc parallel loop  
    reduction(+:sum)  
    for(int i=0;i<n;i++) {  
        sum +=  
            xcoefs[i]*ycoefs[i];  
    }  
    return sum;  
}
```

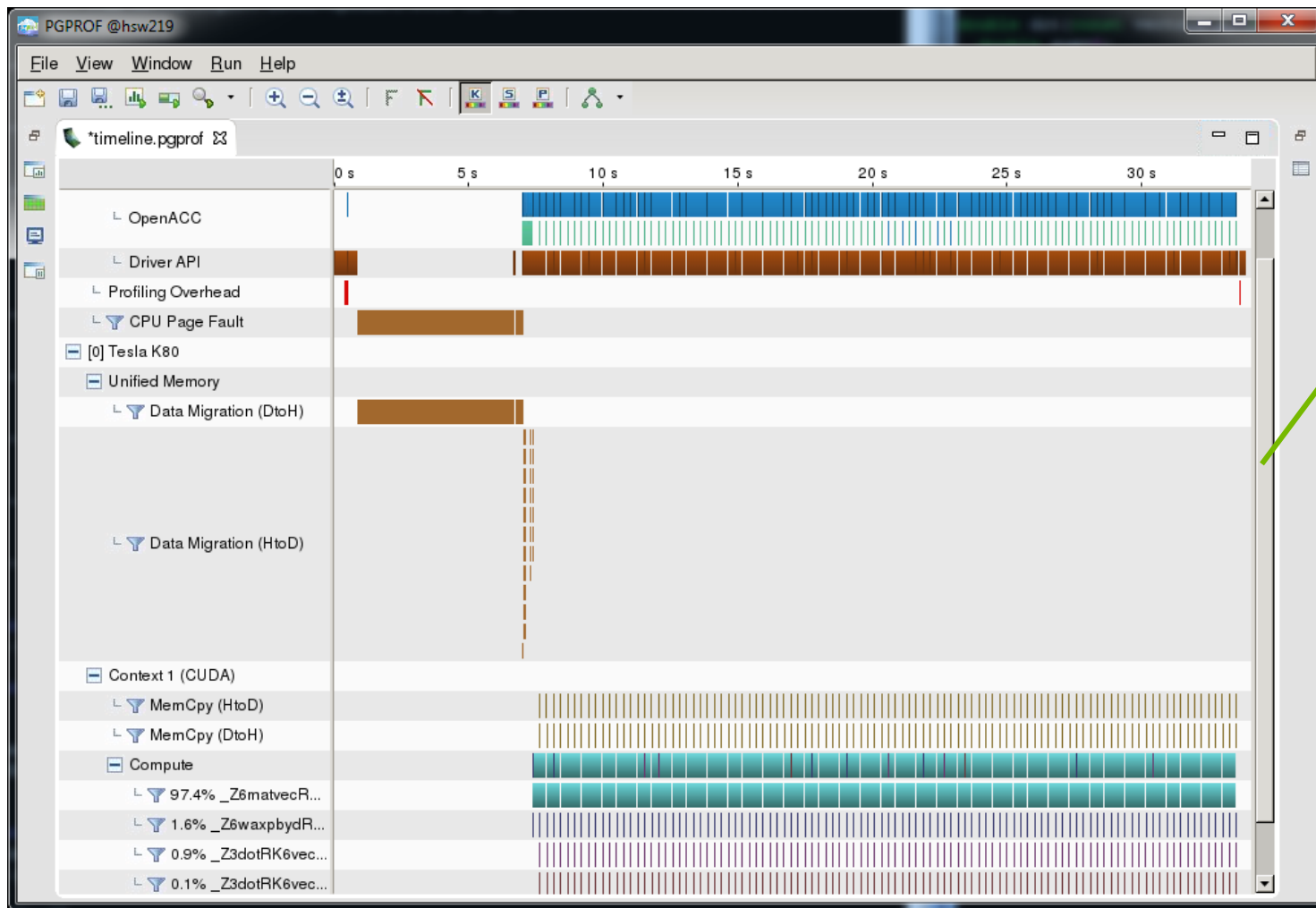
- ▶ Because each iteration of the loop adds to the variable `sum`, we must declare a *reduction*.
- ▶ A parallel reduction may return a slightly different result than a sequential addition due to floating point limitations

Parallelize Matvec

```
void matvec(...) {  
    #pragma acc parallel loop  
    for(int i=0;i<num_rows;i++) {  
        double sum=0;  
        int row_start=row_offsets[i];  
        int row_end=row_offsets[i+1];  
        #pragma acc loop reduction(+:sum)  
        for(int  
j=row_start;j<row_end;j++) {  
            unsigned int Acol=cols[j];  
            double Acoef=Acoefs[j];  
            double xcoef=xcoefs[Acol];  
            sum+=Acoef*xcoef;  
        }  
        ycoefs[i]=sum;  
    }  
}
```

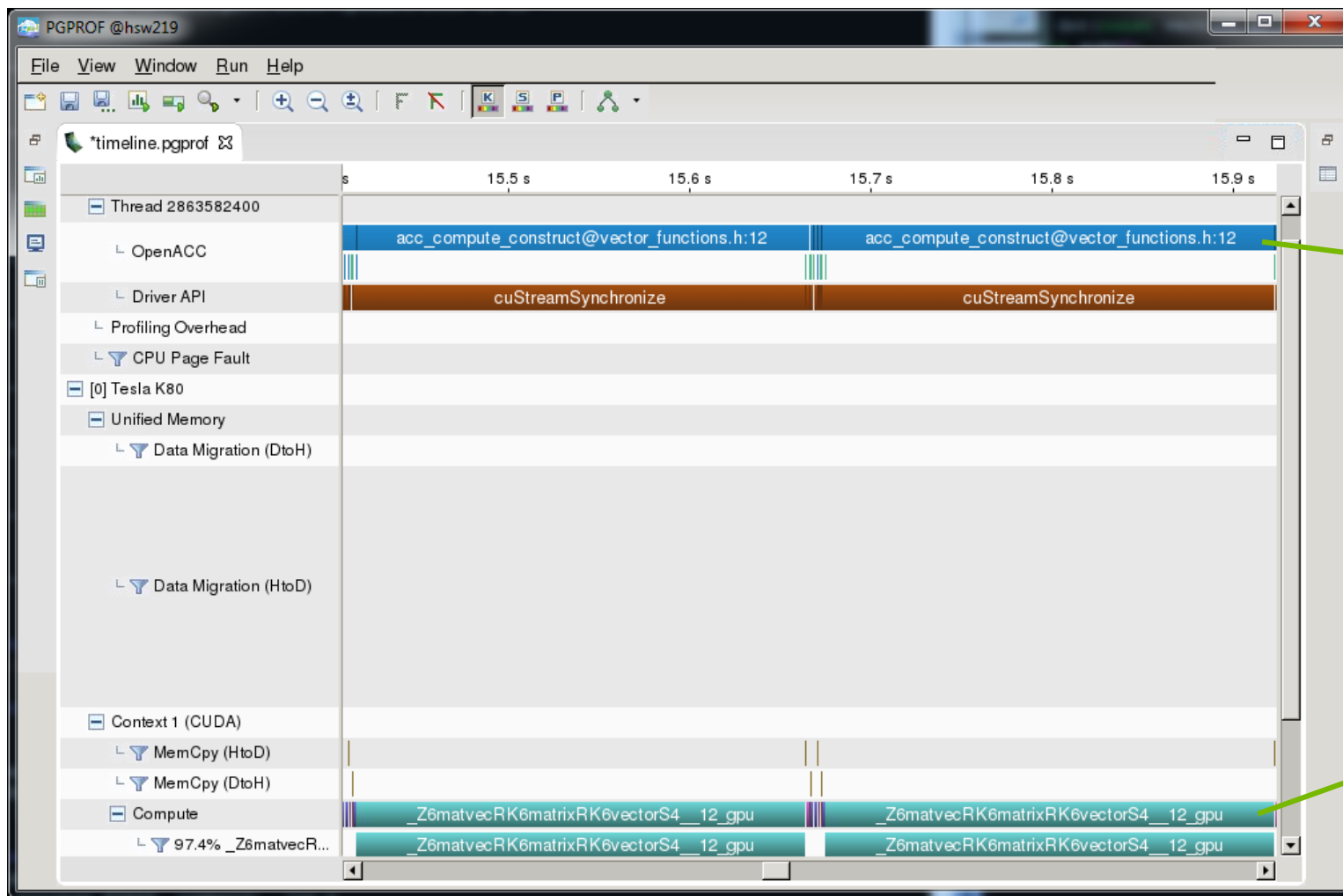
- ▶ The outer *parallel loop* generates parallelism and parallelizes the “*i*” loop.
- ▶ The inner *loop* declares the iterations of “*j*” independent and the reduction on “*sum*”

Final PGPROF Profile for Lecture 1



Now data migration
has been eliminated
during the
computation.

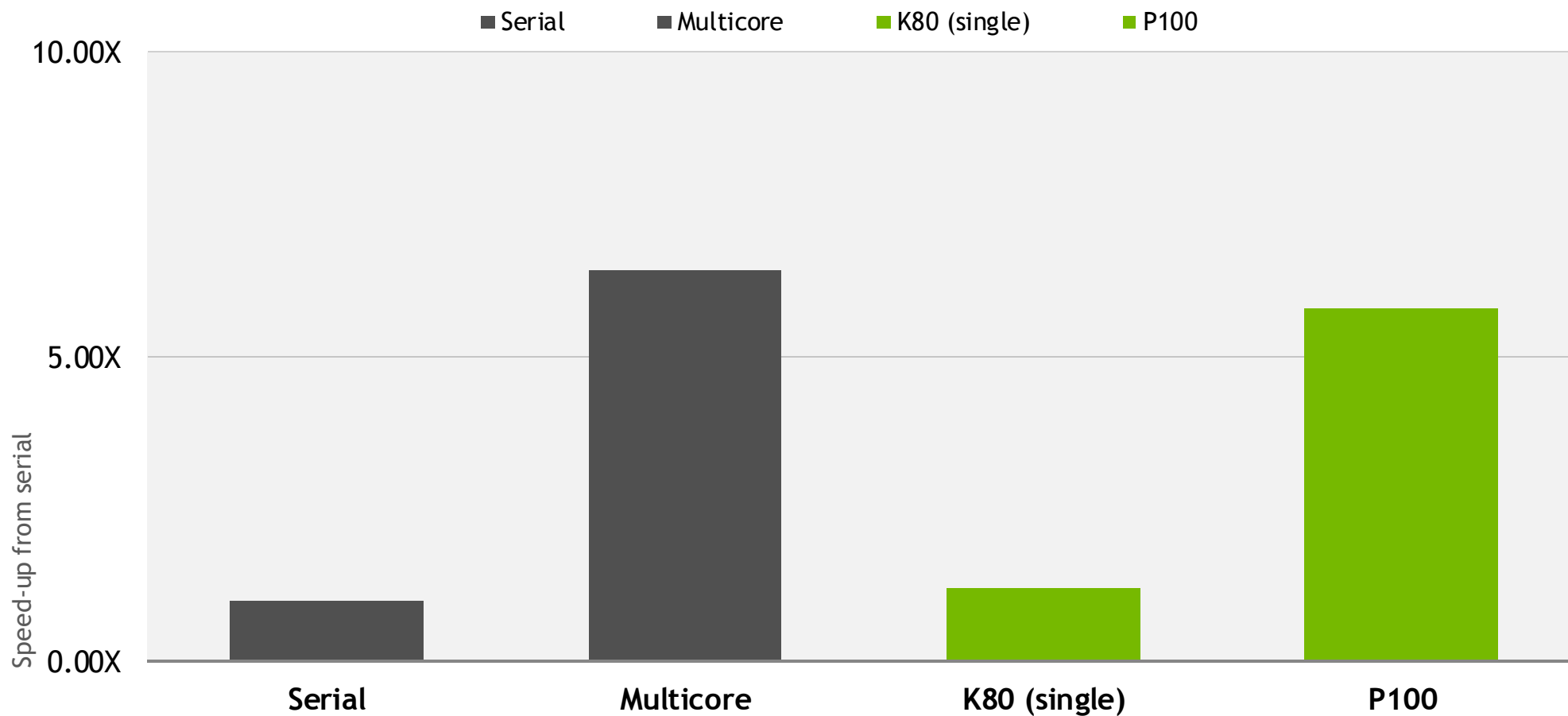
OpenACC Profiling in PGPROF



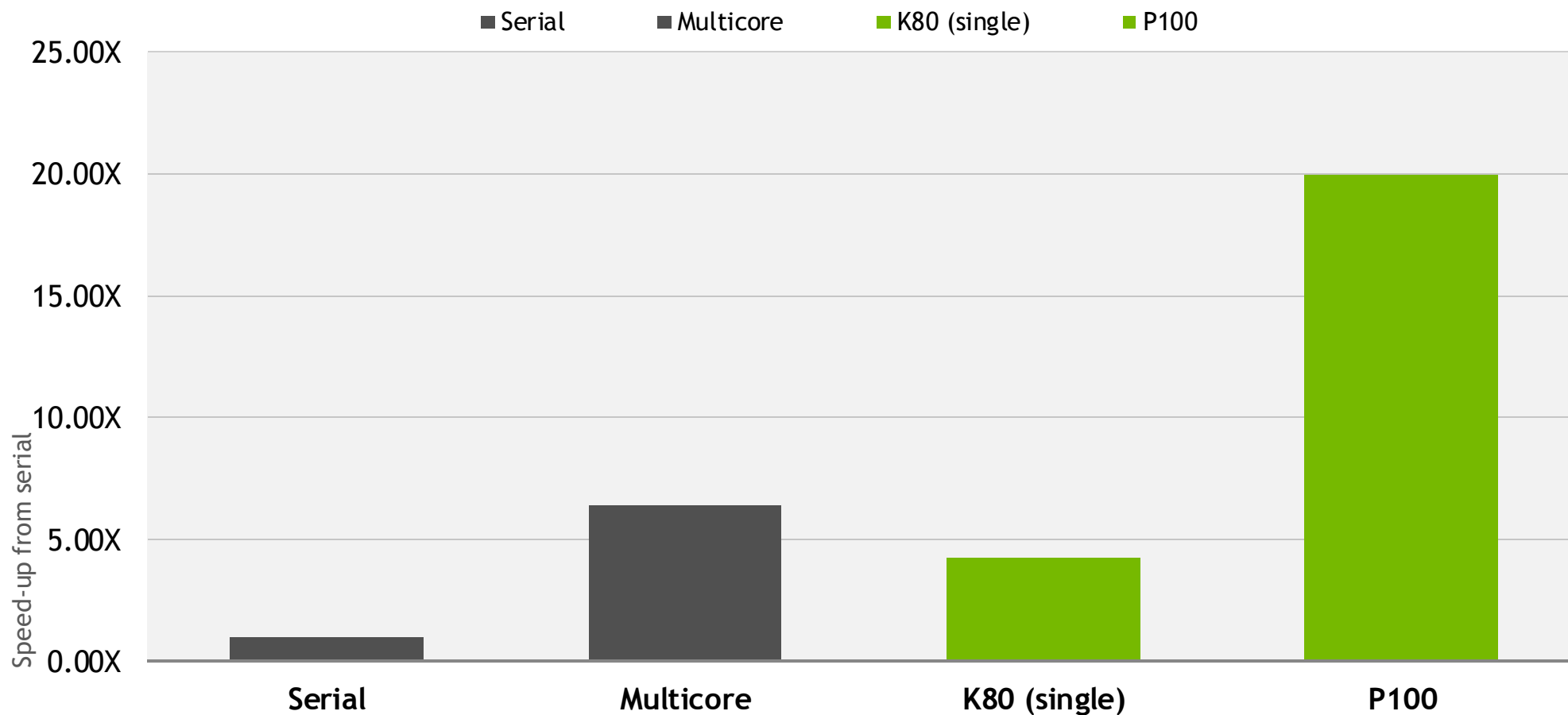
PGPROF will show you where in your code to find an OpenACC region.

We'll optimize this loop next week!

OpenACC Performance So Far...



Where we're going next week...



Optimize (Next Week)

Optimize (Next Week)

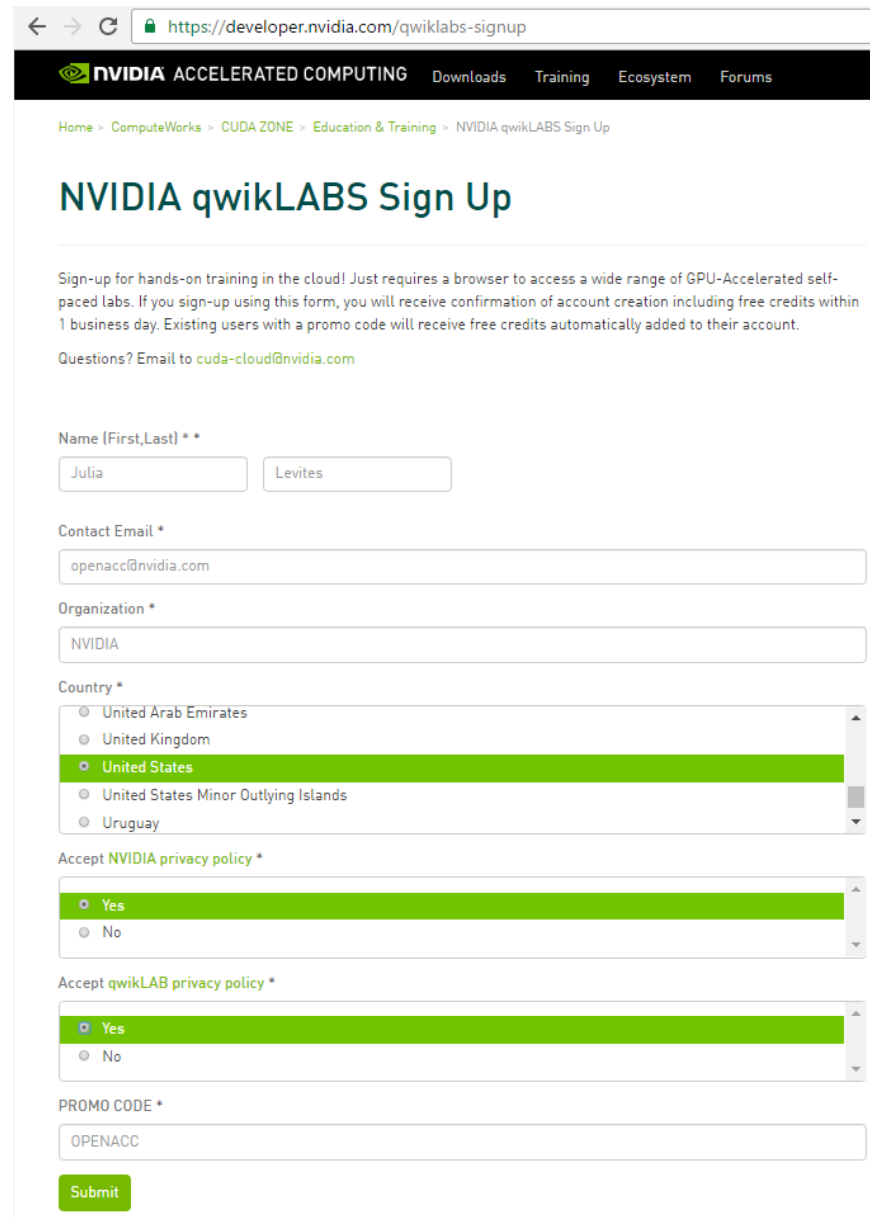
- ▶ Get new performance data from parallel execution
- ▶ Remove unnecessary data transfer to/from GPU
- ▶ Guide the compiler to better loop decomposition
- ▶ Refactor the code to make it more parallel



Using QwikLabs

Getting access

1. Create an account with NVIDIA qwikLABS <https://developer.nvidia.com/qwiklabs-signup>
2. Enter a promo code OPENACC before submitting the form
3. Free credits will be added to your account
4. Start using OpenACC!



The screenshot shows the NVIDIA qwikLABS Sign Up page. The browser address bar displays <https://developer.nvidia.com/qwiklabs-signup>. The page header includes the NVIDIA logo and navigation links: ACCELERATED COMPUTING, Downloads, Training, Ecosystem, and Forums. A breadcrumb trail reads: Home > ComputeWorks > CUDA ZONE > Education & Training > NVIDIA qwikLABS Sign Up.

NVIDIA qwikLABS Sign Up

Sign-up for hands-on training in the cloud! Just requires a browser to access a wide range of GPU-Accelerated self-paced labs. If you sign-up using this form, you will receive confirmation of account creation including free credits within 1 business day. Existing users with a promo code will receive free credits automatically added to their account.

Questions? Email to cuda-cloud@nvidia.com

Name (First,Last) * *

Julia Levites

Contact Email *

openacc@nvidia.com

Organization *

NVIDIA

Country *

- United Arab Emirates
- United Kingdom
- United States**
- United States Minor Outlying Islands
- Uruguay

Accept NVIDIA privacy policy *

- Yes**
- No

Accept qwikLAB privacy policy *

- Yes**
- No

PROMO CODE *

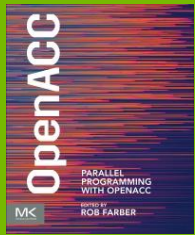
OPENACC

Submit

CERTIFICATION

Available after November 9th

1. Attend live lectures
2. Complete the test
3. Enter for a chance to win a Titan X or an OpenACC Book



Official rules:

<http://developer.download.nvidia.com/compute/OpenACC-Toolkit/docs/TITANX-GIVEAWAY-OPENACC-Official-Rules-2016.pdf>

OPENACC TOOLKIT

Free for Academia

Download link:

<https://developer.nvidia.com/openacc-toolkit>

NEW OPENACC BOOK

Parallel Programming with OpenACC

Available starting Nov 1st, 2016:

<http://store.elsevier.com/Parallel-Programming-with-OpenACC/Rob-Farber/isbn-9780124103979/>

Where to find help

- OpenACC Course Recordings - <https://developer.nvidia.com/openacc-courses>
- PGI Website - <http://www.pgroup.com/resources>
- OpenACC on StackOverflow - <http://stackoverflow.com/questions/tagged/openacc>
- OpenACC Toolkit - <http://developer.nvidia.com/openacc-toolkit>
- Parallel Forall Blog - <http://devblogs.nvidia.com/parallelforall/>
- GPU Technology Conference - <http://www.gputechconf.com/>
- OpenACC Website - <http://openacc.org/>

Questions? Email openacc@nvidia.com

Course Syllabus

Oct 26: Analyzing and Parallelizing with OpenACC

Nov 2: OpenACC Optimizations

Nov 9: Advanced OpenACC

Recordings:

<https://developer.nvidia.com/intro-to-openacc-course-2016>

Questions? Email openacc@nvidia.com

Additional Material

OpenACC kernels Directive

Identifies a region of code where I think the compiler can turn *loops* into *kernels*

```
#pragma acc kernels
```

```
{  
for(int i=0; i<N; i++)  
{  
    x[i] = 1.0;  
    y[i] = 2.0;  
}
```

} kernel 1

```
for(int i=0; i<N; i++)  
{  
    y[i] = a*x[i] + y[i];  
}  
}
```

} kernel 2

The compiler identifies
2 parallel loops and
generates 2 kernels.

Loops vs. Kernels

```
for (int i = 0; i < 16384; i++)  
{  
    C[i] = A[i] + B[i];  
}
```

```
function loopBody(A, B, C, i)  
{  
    C[i] = A[i] + B[i];  
}
```

Loops vs. Kernels

```
for (int i = 0; i < 16384; i++)  
{  
    C[i] = A[i] + B[i];  
}
```

Calculate 0 -16383 in order.

```
function loopBody(A, B, C, i)  
{  
    C[i] = A[i] + B[i];  
}
```

Loops vs. Kernels

```
for (int i = 0; i < 16384; i++)  
{  
    C[i] = A[i] + B[i];  
}
```

Calculate 0 -16383 in order.

```
function loopBody(A, B, C, i)  
{  
    C[i] = A[i] + B[i];  
}
```

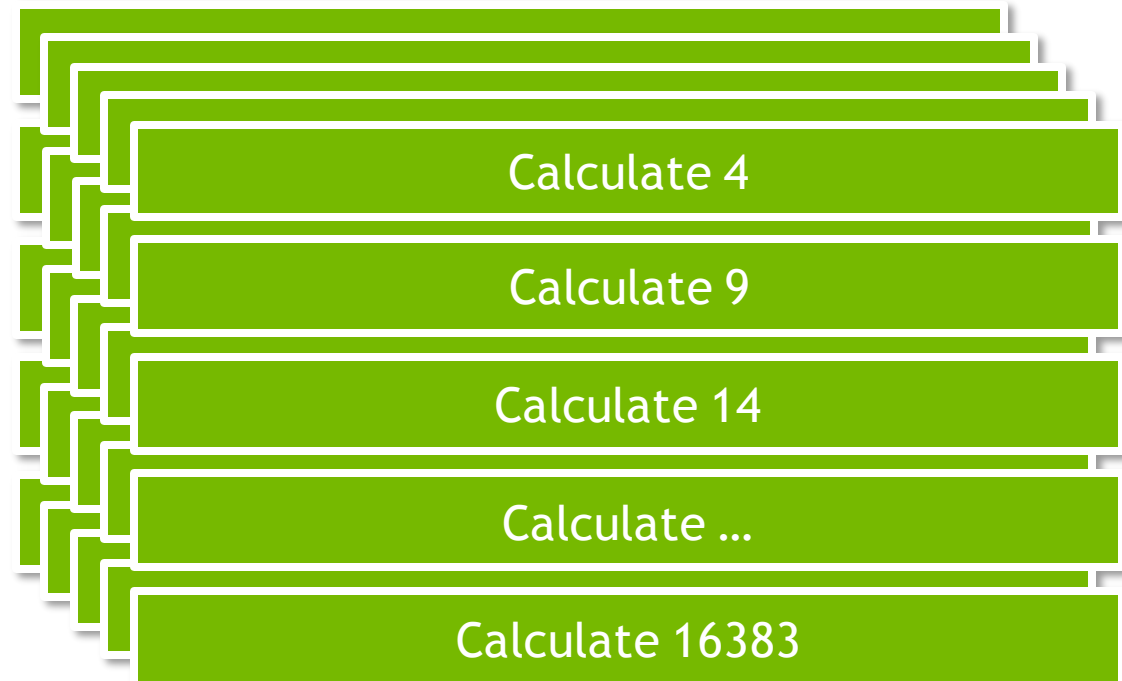
Calculate 0

Loops vs. Kernels

```
for (int i = 0; i < 16384; i++)  
{  
    C[i] = A[i] + B[i];  
}
```

Calculate 0 -16383 in order.

```
function loopBody(A, B, C, i)  
{  
    C[i] = A[i] + B[i];  
}
```



Parallelize Matvec with kernels

```
void matvec(...) {  
    double *restrict ycoefs=y.coefs;  
    #pragma acc kernels  
        for(int i=0;i<num_rows;i++) {  
            double sum=0;  
            int row_start=row_offsets[i];  
            int row_end=row_offsets[i+1];  
            #pragma acc loop reduction(+:sum)  
                for(int  
j=row_start;j<row_end;j++) {  
                    unsigned int Acol=cols[j];  
                    double Acoef=Acoefs[j];  
                    double xcoef=xcoefs[Acol];  
                    sum+=Acoef*xcoef;  
                }  
            ycoefs[i]=sum;  
        }  
}
```

- ▶ With the *kernels* directive, the compiler will detect a (false) data dependency on ycoefs.
- ▶ It's necessary to either mark the loop as *independent* or add the *restrict* keyword to get parallelization.

OpenACC parallel loop vs. kernels

PARALLEL LOOP

- ▶ Programmer's responsibility to ensure safe parallelism
- ▶ Will parallelize what a compiler may miss
- ▶ Straightforward path from OpenMP

KERNELS

- ▶ Compiler's responsibility to analyze the code and parallelize what is safe.
- ▶ Can cover larger area of code with single directive
- ▶ Gives compiler additional leeway to optimize.
- ▶ Compiler sometimes gets it wrong.

Both approaches are equally valid and can perform equally well.

OpenACC Performance So Far... (kernels)

